UNIVERSITÉ DE MONTRÉAL

VÉRIFICATION À LA VOLÉE DE CONTRAINTES OCL ÉTENDUES SUR DES MODÈLES UML

RAVECA-MARIA OARGA
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2005

Your file Votre référence
ISBN: 978-0-494-16825-7
Our file Notre référence
ISBN: 978-0-494-16825-7

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

VÉRIFICATION À LA VOLÉE DE CONTRAINTES OCL ÉTENDUES SUR DES MODÈLES UML

présenté par: OARGA Raveca-Maria

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

Mme BOUCHENEB Hanifa, Doctorat, présidente

M. MULLINS John, Ph.D., membre et directeur de recherche

M. AMYOT Daniel, Ph.D., membre

à Hamadou

REMERCIEMENTS

Tout d'abord je voudrais remercier mon directeur de recherche, le professeur John Mullins qui m'a encouragé à démarrer cette maîtrise, m'a insufflé le goût de la recherche et m'a supervisé tout au long de ces deux dernieres années.

J'aimerais citer et remercier mes professeurs de l'École Polytechnique: H. Boucheneb, N. Ignat, J. Pesant, et S. Wolf, qui par leurs cours et leurs travaux pratiques ont contribué à combler plusieurs lacunes de ma formation. Merci aussi à monsieur Daniel Amyot et à madame Hanifa Boucheneb pour leur participation au jury de ce mémoire.

En ce qui concerne le projet SOCLe qui a encadré ma recherche de maîtrise, je voudrais remercier tout spécialement mon collègue M. Bergeron, premier auteur de la première version de SOCLe, qui m'a beaucoup aidé à m'intégrer au projet et à démarrer mon travail.

Je ne saurais oublier la belle collaboration avec RDDC Valcartier (Recherche et Développement pour la Défense du Canada) et surtout avec M. Robert Charpentier, qui a su nous encourager et conseiller à plusieurs reprises notamment pour faire la présentation du projet chez le IBM et pour la participation au concours des OCTAS (OCTets AS, concours organisé par la FIQ - Réseau de technologies de l'information et des communications de Québec), d'où nous sommes sortis lauréats 2005 dans la catégorie Relève Universitaire.

Finalement, je tiens à remercier tous mes collègues du CRAC de cette époque *André - Aiseinstadt*: D. Azambre, P. Bachand, A. Fellah, S. Lafrance et H. Sardaouna pour leur soutien et le fructueux travail d'équipe, mais aussi pour leurs franche et précieuse camaraderie.

RÉSUMÉ

L'outil SOCLe, conçu au laboratoire CRAC, permet de valider des logiciels orientés-objet, dès la phase de conception, par model-checking de contraintes OCL étendues sur des modèles UML. Ces contraintes disposent des opérateurs de points fixes capables de prendre en compte l'aspect temporel. La première version de l'outil SOCLe exige la construction explicite de l'espace d'états du système à analyser, ce qui limite la taille du graphe à explorer. De plus la création dynamique d'objets UML peut engendrer des espaces d'états infinis qui ne sont pas supportés par le model-checking.

L'objectif de ce mémoire consiste à améliorer l'outil SOCLe en intégrant un algorithme de vérification à la volée. Cette approche, doublée d'une gestion judicieuse de l'espace mémoire, offre l'avantage d'avoir à chaque instant seulement une partie du graphe en mémoire. Cette partie du graphe est construite et guidée en fonction des besoins de vérifications. Elle offre ainsi l'avantage, dans plusieurs circonstances, de contourner le problème d'explosion combinatoire, un handicap majeur du model-checking. En effet elle permet d'arrêter la vérification dès qu'une erreur a été détectée.

L'application d'une telle technique requiert la définition d'un modèle formel du système à vérifier et d'une logique temporelle pour exprimer les propriétés requises. Le modèle formel que nous définissons dans ce travail est une structure de Kripke étendue, capable d'exprimer les notions orientées-objet d'UML. La logique adoptée est une extension d'OCL qui permet d'exprimer d'une part les notions orientées-objet qui le lie à UML, et d'autre part, les propriétés sur les chemins d'exécutions.

Finalement, nous avons validé cette technique par une importante étude de cas proposée par notre collaborateur du RDDC Valcatier. Cette étude de cas modélise et vérifie un système constitué de documents hautement sécurisés. Des tests de comparaisons ont étés effectués sur plusieurs exemples et ont démontré une réduction

du temps de calcul de l'ordre de 30% comparé à la version précédente.

ABSTRACT

The SOCLe tool, conceived at CRAC laboratory, allows the validation of object oriented software, by model-checking the extended OCL constraints on UML models, during the phase of design. Those constraints have fixed points operators that add a temporal consideration.

The early version of the tool's verification requires the construction of the explicit space states of the analyzed system which limits the size of the graph to explore. Moreover the dynamic creation of UML objects may generate infinite state spaces that are not supported by model-checking.

The objective of this thesis is to improve the SOCLe tool by integrating an *on-the-fly* verification algorithm. This approach, doubled by a judicious management of the memory space, offers the advantage of having, at every instant, only one part of the model in memory. This part of the graph is built and conducted on the verification needs. Thus, in many circumstances, it offers the advantage of overcoming the problem of combinatorial explosion, a major problem in model-checking. Indeed, it stops the verification once an error is detected.

The application of such a technique requires the definition of a formal model of the system to verify and the definition of a temporal logic to express its properties. The formal model we defined is an extended Kripke structure able to specify the object-oriented notions of UML. Our logic is an extension of OCL that allows to state on one hand, the object oriented notions that bind it with UML, and on the other hand, the properties of the execution path.

Finally, we validate this technique by an important case study proposed by RDDC Valcartier collaborators. This case study models and verifies a highly secured documents system named Caveats.

Comparison tests that have been done on many examples showed a reduction of about 30% in the computation time compared to the previous version.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS

ASM	Abstract State Machine
CRAC	Conception et Réalisation des Applications Complexes
CTL	Computation Tree Logic
LMCO	Local Model Checking for OCL^{EXT}
LTL	Linear Temporal Logic
OCL	Object Constrant Language
OCL^{EXT}	Extension temporelle d'OCL
OMG	Objects Management Group
RDDC	Recherche et Développement pour la Défense du Canada
SOCLe	Secur OCL environnement
UML	Unified Modeling Language
μ-calcul	Logique temporelle de points fixes

INDEX DES NOTATIONS

LISTE DES TABLEAUX

LISTE DES ANNEXES

CHAPITRE 1

INTRODUCTION

1.1 Motivation

Les ordinateurs représentent une part intégrante dans notre société et dans notre économie, et leurs logiciels deviennent de plus en plus gros et complexes. Le développement de logiciels de qualité est une activité difficile, et passe par le renforcement des activités de conception, de validation et de test. La plupart du temps, le comportement d'un logiciel est validé après l'implantation. Cette technique est coûteuse et parfois peu réalisable.

Pour réduire la complexité du développement de logiciel, la construction du *modèle logiciel* dès la phase de conception, est envisagée. Un *modèle* représente une simplification de la réalité qui ne retient que les éléments pertinents du problème. Le modèle logiciel ne se soucie pas du détail, il ne représente que les besoins les plus importants de la conception. Par analogie, on peut penser à la conception d'un plan de maison. On ne se demande pas comment les briques seront placées dans un mur, mais plutôt si les murs ne s'écrouleront pas, une fois montés.

La description d'un modèle logiciel requiert l'utilisation d'un langage de modélisation. Parmi les langages de modélisation existants, le Unified Modeling Language (UML) (OMG, 2003a) est devenu un standard *de facto* dans l'industrie actuelle. UML est un langage graphique et les logiciels sont représentés visuellement par des diagrammes. Le modèle sous cette forme est assez intuitif, mais il peut porter à confusion entre la phase de conception et d'implantation, car il est possible d'interpréter de plusieurs façons un même diagramme, et il y a aussi le problème de

cohérence entre les différents diagrammes. Pour éliminer les ambiguïtés dans cette phase de développement, un modèle *formel* (mathématique) s'avère nécessaire. Un modèle formel suppose une sémantique formelle pour UML. Malheureusement, UML dans son entité présente seulement une sémantique en langue naturelle. Chaque développeur d'outil d'aide à la conception prend en compte un fragment d'UML et en donne une sémantique formelle.

Le modèle formel ajoute un autre avantage; il permet l'application de méthodes formelles de vérification qui permettent de découvrir des erreurs à ce stade du développement, et donc de réduire le coût de production. Parmi les méthodes formelles, une de plus employée à l'heure actuelle c'est le *model-checking*. L'idée est d'explorer exhaustivement l'espace d'états du système et de vérifier si une propriété est vérifiée ou non. L'utilisation de cette technique d'analyse automatique nécessite des modèles finis en entrée et son plus grand handicap est le problème d'*explosion combinatoire*. On nomme explosion combinatoire le fait qu'un petit changement du nombre de données à considérer dans un problème provoque une explosion du nombre d'états à considérer. Cela peut suffire à rendre sa solution très difficile, voire impossible dans certains cas avec les ordinateurs actuels. Or, dans la modélisation orienté-objet (UML notamment) où les objets peuvent être créés et détruits dynamiquement, l'explosion combinatoire peut provoquer un espace infini d'états, donc créer un réel problème pour l'application du model-checking.

L'outil SOCLe (Secure OCL environnement) développé au laboratoire CRAC (Conception et Réalisation des Applications Complexes), permet de valider des logiciels dès la phase de conception, par le model-checking de contraintes OCL (Object Constraint Language)(OCL 1.1, 1997; OMG, 2002), le langage de spécification de propriétés, sur des modèles UML, le langage de spécification du système.

Dans le but d'améliorer l'outil SOCLe, l'équipe du CRAC, explore présentement

plusieurs méthodes d'abstraction et de réduction de modèles UML. Parmi les méthodes de réduction en vue, la plus avancée est l'analyse statique par *slicing* qui consiste à éliminer toute partie du calcul qui n'influence pas la vérification recherchée.

Une façon de combattre le problème d'explosion combinatoire est d'appliquer une technique alternative, qui combine la simulation avec la vérification et qui permet d'explorer un espace d'états plus volumineux en ne gardant en mémoire qu'une partie du graphe d'exécution piloté par la propriété à vérifier. De plus, une vérification s'arrête dès que la propriété est vérifiée ou falsifiée. Si cette propriété est trouvée fause, un contre-exemple est produit pour aider le concepteur à corriger les erreurs. Elle permet également de réduire le temps de calcul dans plusieurs cas significatifs. Il s'agit de la vérification *à la volée* (*on-the-fly*), technique adoptée et analysée dans ce mémoire qui vise en grande partie l'amélioration de l'outil SOCLe au niveau de la vérification. Une telle technique requiert un modèle formel orienté UML et une logique temporelle qui intègre la spécification d'OCL. La validation de SOCLe avec cette technique est réalisée par des études de cas.

1.2 À propos de Socle

1.2.1 Concepts employés par Socle

Le projet SOCLe a développé une sémantique formelle pour un fragment significatif d'UML qui intègre les expressions OCL, la création d'objets et l'héritage comportemental. De plus, l'approche permet la vérification des contraintes OCL, via les techniques de model-checking. Les trois paragraphes suivants donnent une synthèse succincte des trois concepts utilisés par l'outil SOCLe: UML, OCL et model-checking.

UML Unified Modeling Language est né de la fusion de trois méthodes qui ont influencé la modélisation orientée-objet au milieu des années 90: OMT (Rumbaugh et al., 1991), Booch (Booch, 1994) et OOSE (Jacobsen et al., 1992). En l'espace de quelques années seulement (1997), UML est devenu une norme incontournable de l'OMG (Object Management Group). Le langage UML comble une lacune importante des technologies orientées-objet. Il permet d'exprimer et d'élaborer des modèles orientés-objet, indépendamment de tout langage de programmation. Il a été conçu pour servir de support à une analyse basée sur les concepts orientés-objet. Sa notation graphique permet d'exprimer visuellement une solution orientée-objet, ce qui facilite la comparaison et l'évaluation de plusieurs solutions. Finalement, la véritable force d'UML est qu'il repose sur un métamodèle, ce qui permet de classer les différents concepts du langage (selon leur niveau d'abstraction ou leur domaine d'application) et d'exposer ainsi clairement sa structure. Un métamodèle décrit d'une manière très précise tous les éléments de la modélisation (les concepts véhiculés et manipulés par le langage) et la syntaxe de ces éléments (leur définition et le sens de leur utilisation). Donc, au niveau du métamodèle, UML possède une syntaxe formelle. On peut noter que le métamodèle d'UML est lui-même décrit de manière standardisée par un méta-métamodèle, à l'aide de MOF (Meta Object Facility), une norme OMG (Object Management Group) de description des métamodèles. Par contre, une faiblesse d'UML est qu'il ne définit pas le processus d'élaboration des modèles.

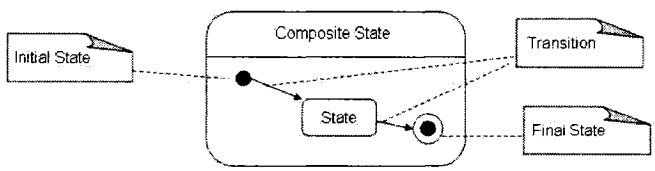

Figure 1.1 Syntaxe concrète d'un diagramme d'états-transitions

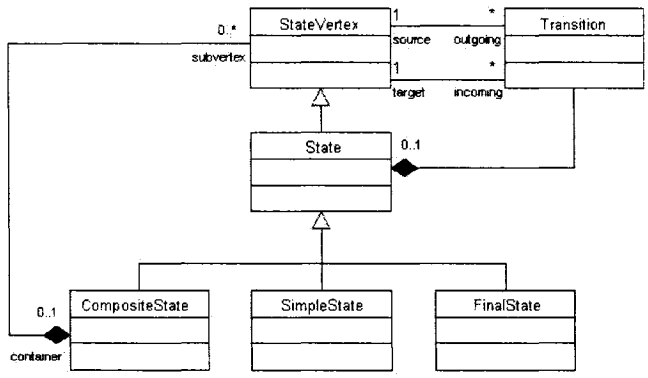

Figure 1.2 Syntaxe abstraite d'un diagramme d'états-transitions

UML est décrit par deux éléments: la syntaxe abstraite, la syntaxe concrète. Le métamodèle décrit ce qui est valide dans un modèle UML. Il énumère tous les éléments du modèle, leurs attributs et leurs associations avec d'autres éléments du modèle. Il n'utilise que le diagramme de classes (décrit en détail au chapitre 3, section 3.1) et les contraintes OCL (voir chapitre 4).

La figure 1.1 illustre un exemple de la syntaxe concrète d'un diagramme d'états-transitions et la figure 1.2, la syntaxe abstraite correspondante[1]. Ce diagramme définit un état comme un type spécial d'un état sommet (vertex). Chaque état sommet peut avoir plusieurs transitions entrantes et sortantes. Les états peuvent avoir des transition internes. Un état peut être de plusieurs types: simple, composite et final. Un état composé peut contenir d'autres états (composés, simples ou finaux). C'est ainsi que sont exprimés les diagrammes UML à partir de méta-éléments.

La spécification d'un modèle en UML est faite par un ensemble de diagrammes. La vue statique du système est captée par les diagrammes d'objets, de classes, de composants et de déploiement, et la vue dynamique par les diagrammes d'états, de

[1]Exprimée à l'aide d'un diagramme de classes

collaboration, de séquence et d'activités. La signification des diagrammes UML les plus utilisés est:

- le *diagramme d'utilisation* est concentré sur les interactions entre l'usager du système (nommé *acteur*) et le système lui même.
- le *diagramme de classes* décrit la structure statique du système et les relations entre les classes.
- le *diagramme de séquences* décrit le comportement du système en regardant les interactions entre les objets et les messages qu'ils s'échangent.
- le *diagramme d'objets* modélise un seul état du système, parfois nommé *snapshot*. Il contient les objets avec les valeurs de leurs attributs et les liens entre les objets.
- le *diagramme d'états-transitions* décrit le cycle de vie d'un objet sur une séquence d'états, par une machine à états, attaché à chaque objet.

OCL Object Constraint Language (OMG, 2002; OCL 1.1, 1997) est le langage de spécification des contraintes d'UML. OCL possède une syntaxe concrète qui permet d'exprimer des expressions et des contraintes. Les expressions OCL sont employées dans les digrammes UML pour éliminer certaines ambiguïtés et pour augmenter l'expressivité d'UML:

- spécifier la valeur initiale d'un attribut ou d'une association;
- spécifier des affectations;
- spécifier le corps d'une méthode;

- indiquer une instance dans un diagramme dynamique (ex: *self* sur un contexte (classe) représente l'objet courant du contexte, donc l'instance d'une classe);
- indiquer une condition dans un diagramme dynamique (ex: *guard* dans un diagramme d'états-transitions ou de séquence);
- indiquer les valeurs locales des paramètres formels d'une méthode;

Les expressions OCL sont aussi utilisées comme langage de navigation dans certains diagrammes UML (diagramme de classes, diagramme d'objets).

Les contraintes OCL spécifient des conditions que le système doit respecter. Il y a deux types des contraintes: les invariants et les pré/post-conditions. Un invariant est une contrainte qui porte sur toutes les instances d'une classe. Il s'agit d'une condition supplémentaire qui s'applique à une classe, un type ou une association, et qui est impossible à spécifier en utilisant la notation graphique d'un modèle UML. Une pré/post-condition est une contrainte qui définit un contrat que le comportement d'une méthode doit satisfaire. La pré-condition décrit la condition qui doit être respectée avant l'exécution de la méthode. La post-condition décrit l'effet produit par l'exécution de la méthode. Une contrainte OCL est exprimée sous la forme d'un patron qui contient une expression OCL (dans le cas d'un invariant) ou deux expressions OCL (dans le cas de pré/post-condition), et un contexte bien précis.

Model-checking Le *model-checking* est une technique de vérification automatique des systèmes informatiques (logiciels, circuits logiques, protocoles de communication), représentés par des modèles formels. Il s'agit de tester algorithmiquement si un modèle donné, le système lui-même ou une abstraction du système, satisfait une propriété, généralement formulée en termes de logique temporelle.

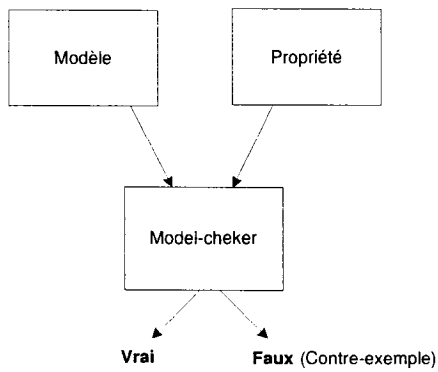

Figure 1.3 Model-checker

Pour effectuer du model-checking nous avons besoin d'exprimer le système sous un modèle opérationnel: automates comme les automates de Büchi (Büchi, 1960), système de transition comme la structure de Kripke (S.A. Kripke, 1963), etc. Les propriétés requises sont exprimées dans un langage souvent appelé logique. Parmi les logiques les plus connues citons: la logique temporelle linéaire LTL (Linear Temporal Logic) (Pnueli, 1997), la logique arborescente CTL (Computation Tree Logic) (Clarke et al., 1983), CTL* qui présente la force d'expression du μ-calcul (Kozen, 1983), une logique qui se base sur les points fixes.

Les deux classes de propriétés les plus vérifiées par un model-checker sont:

- *la sûreté*, stipulant que quelque chose de mauvais n'arrive jamais, comme par exemple la violation d'un invariant.
- *la vivacité*, stipulant que quelque chose de bon est toujours possible, comme par exemple la terminaison dans le cas des programmes séquentiels.

Le model-checking est dit *global* quand la propriété doit être vérifiée dans chaque état du système ou *local* quand la formule doit être vérifiée dans un état spécifique, qui est généralement l'état initial.

L'outil qui travaille avec cette technique s'appelle *model-checker*. Il contient l'implantation des algorithmes de model-checking, l'environnement d'édition et d'exécution et une interface. Le model-checker prend en entrée une représentation du système à vérifier et la spécification de la propriété qu'il doit respecter, et il fournit la réponse **Vrai** si la propriété est vérifiée ou **Faux** avec un contre exemple, dans le cas contraire (figure 1.3). Si une propriété est vérifiée on dit que le système est un modèle de la propriété.

La technique *à la volée* est basée sur l'observation suivante: il n'est pas nécessaire de conserver le graphe entier en mémoire pour vérifier une formule temporelle. De plus, une vérification locale s'arrête dès que la formule est falsifiée. Dans ce cas, un contre-exemple est produit pour aider le concepteur à corriger les erreurs. Souvent, des erreurs sont découvertes très tôt pendant la recherche, ce qui évite l'exploration de l'espace d'états en entier et réduit le coût du développement.

1.2.2 Vue d'ensemble de l'outil SOCLe

L'architecture de SOCLe est illustrée à la figure 1.4 et comporte trois modules:

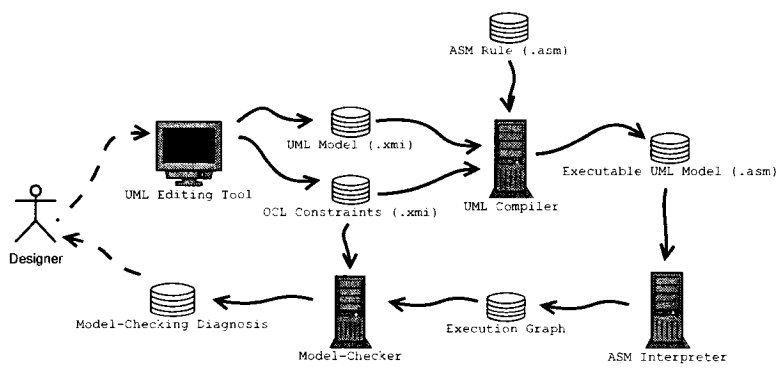

Figure 1.4 Architecture de SOCLe

1. Compilateur `XmiToAsm`: traduit le format `UML` dans le formalisme `ASM` (Ab-

stact State Machine) (Kutter and Pierantonio, 1997).

2. Interpréteur: permet d'exécuter le `ASM` et obtenir le graphe d'exécution.

3. Model-checker: vérifie les contraintes.

L'éditeur employé pour la modélisation `UML` est *ArgoUML*, un logiciel libre (*open-source*) développé par Tigris [2]. Le modèle est exprimé dans XML Metadata Interchange (XMI) un format qui est accepté par la plupart des outils UML. L'outil SOCLe comporte une interface qui est intégrée à ArgoUML.

Modélisation Un modèle `UML` de SOCLe correspond exactement à trois diagrammes: le `diagramme de classes`, le `diagramme d'états-transitions` et le `diagramme d'objets`.

La structure statique du système est spécifiée par le `diagramme de classes`. La figure 1.5 illustre un diagramme de classe qui contient des classes et des associations. Une classe comporte des attributs et des méthodes. Dans un modèle, au moins une classe doit être un *thread* (processus léger). Les threads sont des processus qui représentent l'exécution d'un ensemble d'instructions et qui possèdent une pile d'appels. Plusieurs threads impliquent la concurrence. L'héritage est supporté par l'outil SOCLe et la relation d'héritage est spécifiée comme une relation de sous-typage. La redéfinition des méthodes héritées est possible. Chaque classe peut être instanciée une ou plusieurs fois (une nombre borné), sous forme d'objets. Les objets peuvent être créés dynamiquement.

Le comportement dynamique de chaque classe est modélisé par les `diagrammes d'états-transitions`. L'état initial d'un modèle `UML` est spécifié par le `diagramme`

[2] http://argouml.tigris.org/

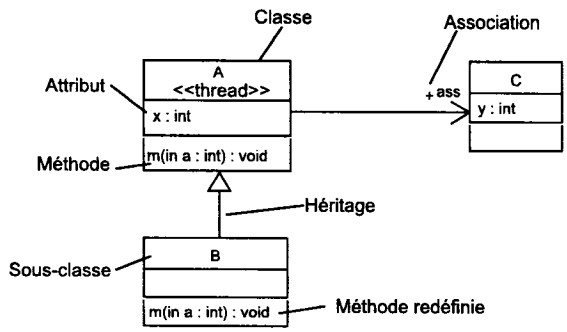

Figure 1.5 Diagramme de classes

d'objets. La description détaillée de la modélisation est développée dans le chapitre 3 section 3.1, dans la mesure où les expressions OCL interviennent.

Les expressions OCL permettent de modéliser plus précisément le comportement dynamique du système. Pour la modélisation elles jouent un rôle plus important dans le diagramme d'états-transition pour spécifier les transitions: garde, actions (voir chapitre 3, section 3.1).

Les contraintes OCL sont exprimées sous forme d'un patron qui indique le contexte et la condition qui doit être respectée, exprimée sous forme d'une expression OCL booléenne. Elles seront explicitement détaillées dans le chapitre 4, section 4.5.

Vérification La vérification des contraintes OCL comporte trois phases. Les deux premières phases sont basées sur la sémantique du modèle UML, exprimée en ASM. Voir (Bergeron, 2004) pour une description détaillée.

Dans la première phase, le modèle UML est traduit par le compilateur dans le formalisme ASM conformément à la sémantique statique d'UML. Dans cette traduction, les éléments de base comme les noms de classes et de méthodes sont associés à des *types* ASM et les éléments plus complexes comme les déclarations de méthodes, les tran-

sitions de diagrammes d'états-transitions sont associées à des *fonctions énumérées.* Le diagramme d'objets aussi, est traduit dans de telles fonctions. On obtient le modèle UML exécutable sous forme ASM.

Dans la deuxième phase, l'interpréteur exécute la spécification ASM. Il prend comme état de départ l'état initial et applique une règle qui capture la sémantique dynamique d'UML. Cette règle est structurée de la sorte:

- Choisir un *thread*, i.e. la pile d'appel courante.
- Sélectioner l'objet au-dessus de la pile d'appel courante, i.e. l'objet qui est en cours d'exécution selon le *thread* choisi.
- Choisir une transition active pour cet objet.
- Exécuter la transition si possible.

La dernière phase correspond à la vérification de contraintes OCL. Les contraintes OCL sont alors traduites dans des μ-formules qui contiennent des expressions OCL considérées comme des propriétés atomiques. L'algorithme de vérification utilisé est celui de (Cleaveland, 1990), appliqué au graphe d'exécution obtenu à la phase précédente. Il faut souligner que pendant l'exécution du graphe, toutes les expressions OCL sont évaluées.

Inconvénients Cette technique de vérification exige la construction complète du graphe d'exécution, ce qui entraîne des inconvénients. Premièrement, il faut garder en mémoire le système de transitions au complet, ce qui limite la taille des modèles analysables, compte tenu que, dans le formalisme ASM un état est représenté par une structure de données complexe: un ensemble de fonctions statiques, dynamiques et externes. Un graphe d'une taille importante, implique un

temps de calcul aussi important et ce, même pour une formule falsifiée au tout début du graphe. L'ASM s'avère donc correcte comme formalisme pour la simulation, mais pour la vérification il n'est pas assez abstrait pour permettre d'isoler les éléments orientés-objet nécessaires, en vue d'explorer des espaces d'états plus importants. Deuxièmement, la création dynamique peut facilement provoquer un espace d'états infini et donc élimine la possibilité de la vérification avec cette technique. Et dernièrement, on peut ajouter le fait que certaines propriétés sont difficiles à exprimer par le μ-calcul (voir l'annexe II.2, section II.2).

1.3 Objectifs

L'objectif principal de ce mémoire est d'améliorer le module de vérification de l'outil SOCLe. Ceci requiert l'accomplissement des sous-objectifs suivants:

- Appliquer une stratégie de vérification alternative qui n'exigera pas d'élaborer au complet le graphe d'exécution. Ceci permettra de:
  - Réduire le temps de calcul.
  - Explorer des modèles plus importants en terme de taille.
- Évaluer de façon empirique les améliorations apportées à l'outil.

1.4 Méthodologie

Dans le but d'atteindre les objectifs indiqués, nous proposons une méthodologie par étapes:

1. À partir de la sémantique ASM de (Bergeron, 2004) on se propose d'extraire un modèle formel abstrait qui ne garde que les informations significatives du

modèle, tout en exprimant les paradigmes orientés-objet (héritage, création d'objet, polymorphisme), et qui, par la suite, nous permet d'exprimer la sémantique de la logique adoptée pour la vérification.

2. Exprimer la syntaxe et la sémantique d'expressions OCL employées dans la modélisation avec l'outil SOCLe.

3. Exprimer la syntaxe et la sémantique de la logique permettant l'expression des types de contrats temporaux plus étendus (invariant, pre/post conditions et autres).

   OCL est un langage de spécification qui permet d'augmenter la précision des diagrammes UML et de vérifier des contraintes statiquement (dans un état donné du système). Pour exprimer des propriétés sur des chemins d'exécution, nous avons deux possibilités: étendre OCL avec des prédicats temporaux pertinents, ou dans une logique temporelle d'inclure les expressions OCL (voir chapitre 2, section 2.4). La logique temporelle adoptée doit être pertinente pour appliquer la vérification à la volée.

   Dans ce mémoire, la logique proposée est une extension d'OCL à deux niveaux. Dans le premier niveau, on retrouve la logique de spécification OCL, enrichie avec des opérateurs spécifiques orientés-objet. Dans le deuxième niveau, on retrouve la logique arborescente CTL où les propriétés atomiques sont remplacées par des expressions OCL booléennes, et dans laquelle les opérateurs universel et existentiel sont ajoutés.

   Ce choix est motivé par le fait que la vérification de formules CTL se prête facilement à une vérification à la volée. L'algorithme de vérification adopté pour l'implantation est une adaptation de (B. Vergauwen and J. Lewi, 1993).

4. L'intégration dans l'outil SOCLe sera faite en plusieurs étapes:

   - Implantation de l'algorithme local naïf (temps d'exécution quadratique,

i.e. $\mathcal{O}(n^2)$)

- Implantation de l'algorithme local linéaire (temps d'exécution linéaire, i.e. $\mathcal{O}(n)$)
- Raffinement de cette implantation. Nous proposons une gestion automatique de l'espace mémoire. Lorsque la limite d'espace mémoire sera dépassée, tous les états qui ne sont plus pertinents à la vérification locale de la formule seront éliminés. Il s'agit ici d'appliquer une technique d'élagage. Dans plusieurs cas significatifs, cette approche permet de contourner efficacement le problème bien connu d'explosion de l'espace d'états.

5. Une importante étude de cas, développé pour le compte de RDDC Valcartier (gouvernement du Canada), nous permettra de valider l'outil. Il s'agit de modéliser et vérifier les *Caveats*, qui représentent des *catégories de sécurité*, qui s'additionnent aux restrictions de classifications sécurisées existantes des documents. Elles présentent une importance pour le Ministère National de la Défense (Departement of National Defense - DND) et leur Système de Contrôle (Command and Control Information Systems - C2IS). Cette étude de cas fait l'objet d'une publication (Painchaud et al., 2005).

1.5 Structure du mémoire

Ce mémoire est organisé comme suit:

Le chapitre 2 présente l'état de l'art, qui situe l'outil SOCLe parmi les outils existants dans la littérature, et présente des logiques étendues qui se prêtent à exprimer les notions orientées-objet et la vérification par model-checking.

Le chapitre 3 fait un rappel sur `UML` et présente le fragment `UML` adopté par SOCLe.

Ensuite, le modèle formel proposé est décrit.

Le chapitre 4 propose un rappel sur OCL standard, exprime la syntaxe et la sémantique des expressions OCL employées dans notre outil, et expose la syntaxe et la sémantique de la logique proposée. La dernière section est dédiée au model-checking à la volée et aux algorithmes implantés.

Dans le chapitre 5 l'outil SOCLe est validé par une étude de cas, et l'efficacité de la technique alternative adoptée est discutée.

Le chapitre 6 conclut le mémoire avec une synthèse résumant les principaux éléments du mémoire, présente un aperçu des travaux futurs et une réflexion finale.

CHAPITRE 2

ÉTAT DE L'ART

Ce chapitre présente l'état de l'art en rapport avec l'outil SOCLe, la logique temporelle qui exprime les contraintes sur les modèles UML et la méthode de vérification à la volée proposée. La première partie présente des travaux qui appliquent comme approche la vérification à la volée, mais pas sur les modèles UML. La deuxième partie décrit des outils qui utilisent UML mais dans lesquels la spécification des contraintes n'est pas en OCL. Ensuite, on présente des outils basés sur UML et OCL. La dernière partie est consacrée aux logiques temporelles étendues.

2.1 Outils - À la volée

Cette section présente des outils qui utilisent un *model-checker à la volée*, mais ne sont pas spécialement orientés-objet.

2.1.1 RuleBase

Dans (Beer et al., 1998), les auteurs définissent les spécifications du langage RCTL (Regular CTL), une extension de CTL, par des expressions régulières qui sont plus intuitives et augmentent l'expressivité des propriétés. Les résultats de Beer et al. ont été implantés en 1995 dans l'outil RuleBase (Beer et al., 1997), un projet d'IBM basé sur des techniques de vérifications symboliques. RuleBase lit la formule RCTL et décide s'il est plus efficace d'utiliser la vérification *à la volée*. Les formules qui ne

peuvent pas être vérifiées à la volée sont évaluées en utilisant l'algorithme original de CTL. Beer et al. rapportent qu'un nombre important d'erreurs sont détectées par RuleBase en utilisant la vérification à la volée et que cette dernière est très efficace par rapport à la vérification classique de CTL. Dans la plupart des cas, les erreurs détectées à la volée n'ont pu l'être autrement, à cause de l'explosion combinatoire.

2.1.2 SPIN

SPIN (Holzmann, 2003) est employé pour la détection des erreurs dans les systèmes distribués, tels que les systèmes d'exploitation, les protocoles de communication de données, les algorithmes concurrents, etc. La description du système est faite dans le langage de spécification de haut niveau PROMELA (PROcess MEta LAnguage) (Holzmann, 1993), qui permet la modélisation de la concurrence par le non-déterministe.

SPIN peut utiliser comme méthode de vérification un model-checking complet en faisant une recherche exhaustive, ou travailler *à la volée*, i.e. éviter la construction globale du graphe d'exécution comme pré-requis de la vérification. Il construit le graphe d'exécution en fonction de ses besoins, c'est-à-dire en fonction de la formule à vérifier.

À l'étape de la vérification, le système est traduit en un automate. La propriété requise est exprimée dans la logique temporelle LTL (Linear Temporel Logic) (Pnueli, 1997), et c'est la négation de la formule à vérifier qui est traduite en automate de Büchi. Si le langage reconnu par le produit synchronisé des deux automates est vide, alors le système satisfait la formule. Sinon, une séquence appartenant au langage du produit est retournée comme contre-exemple à la spécification.

De plus, cet outil exploite des techniques de réduction d'ordres partiels pour limiter les entrelacements dus à la concurrence et des diagrammes de décision binaire BDD (Binary Decision Diagram) comme structure de données pour optimiser la vérification.

L'expressivité de la logique *LTL* est assez limitée, ce qui peut causer un inconvénient. De plus, pour utiliser SPIN, il faut connaître cette logique pour pouvoir exprimer les propriétés. Il faut signaler que la spécification PROMELA ne supporte pas directement la modélisation orientée-objet.

2.1.3 CADP

Construction and Analysis of Distributed Processes (CADP) (Garavel et al., 2002; Fernandez et al., 1991), connu sur le nom "*CAESAR/ALDEBARAN Development package*", est une boîte à outils pour la conception des protocoles de communication et des systèmes distribués, et permet la compilation, la simulation interactive et guidée, la vérification formelle et à la génération de tests. Il utilise le langage LOTOS (Language of Temporal Ordering Specification) (Bolognesi and Brinksma, 1987), basé sur les algèbres des processus pour le contrôle et les spécifications algébriques pour les données.

La figure 2.1 décrit l'architecture du projet CADP. On y retrouve:

1. Le spécificateur *LOTOS* (un standard ISO).
2. Le compilateur *Caesar.ADT* traduit LOTOS dans un langage qui permet la simulation (via le langage C) et permet aussi d'obtenir un modèle formel, une machine à états (i.e. un système de transitions étiquetées fini).
3. Les vérificateurs par équivalence (bisimulation): *BCH_MIN* (technique de

minimisation) et *ALDEBARAN* (technique de comparaison).

4. Deux model-checkers: *XTL* et *EVALUATOR*.

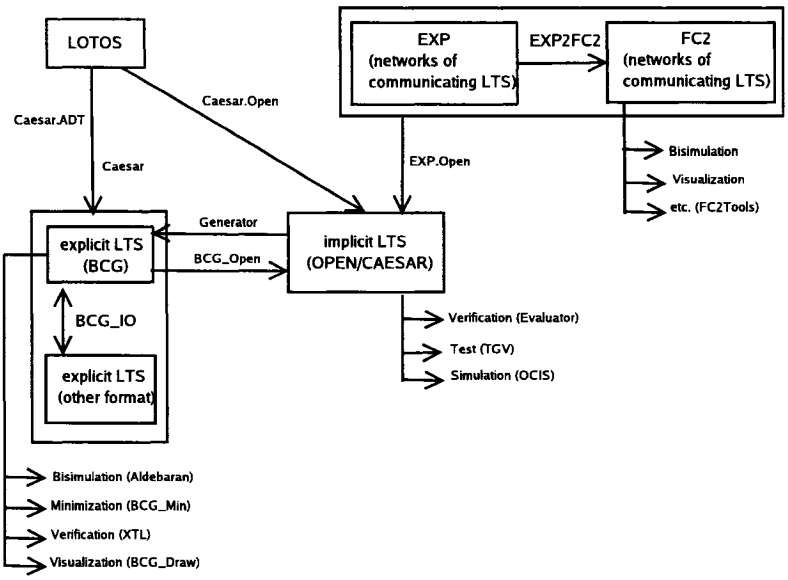

Figure 2.1 Environnement CADP

EVALUATOR est le *model-checker à la volée* employé dans ce projet. En ce qui concerne la logique utilisée, la présentation la plus complète est donnée dans (Mateescu and Sighireanu, 2003). On y présente une logique temporelle et la méthode de vérification associée. L'approche vise un bon compromis entre l'expressivité du formalisme des propriétés, la complexité du problème de vérification et la facilité de l'utilisation de l'interface. La logique est une extension du μ-calcul sans alternance, avec des formules sur les actions comme en ACTL (la logique CTL étendue par des actions) (Nicola and Vaandrager, 1990), et des expressions régulières comme en PDL - Propositional Dynamic Logic (une logique qui permet d'exprimer des interaction de programmes) (Fischer and Ladner, Apr). Cela permet une description concise, intuitive et expressive des propriétés de sûreté et de vivacité des systèmes de transitions étiquetées. La méthode de vérification est basée sur la traduction

du problème vers un système d'équations booléennes qui est résolu au moyen d'un algorithme efficace ayant une bonne complexité moyenne. L'algorithme permet aussi de générer des diagnostics.

2.2 Outils - UML sans OCL

Les trois outils qui seront présentés dans cette catégorie permettent la modélisation en UML, mais pour faire la vérification formelle, une traduction vers le langage d'entrée d'un model-checker adopté est requise.

2.2.1 IF

IF (Bozga et al., 2004; Bozga et al., 2002) est un environnement de modélisation et de validation de systèmes hétérogènes temps-réel: les algèbres de processus et les langages synchrones (qui supposent des interactions atomiques) et les systèmes communicants asynchrones (de modèles en SDL et UML, qui génèrent des interactions non atomiques).

L'outil IF présente trois niveaux de spécification. Dans le premier niveau, on retrouve des primitives temps-réel et des extensions de langages haut niveau de modélisation, comme SDL (Specification and Decription Language) ou UML. Le deuxième niveau correspond à la spécification IF, un langage intermédiaire qui sert de modèle sémantique. Le troisième niveau correspond au modèle formel, qui est un système de transitions étiquetées. Ce modèle formel est employé pour la vérification par model-checking ou la génération des tests.

La figure 2.2 présente une vue d'ensemble de l'architecture de l'outil IFx qui est spécialisé dans la modélisation en UML. Le modèle conçu dans le langage UML est

traduit dans la spécification IF. Il est basé sur des automates temporisés, étendus avec des données, nommés *processus*. Comme dans les *statecharts* de Harel (Harel, 1987), les automates sont hiérarchisés. Une transition est étiquetée par un *trigger* (un déclencheur), un *guard* (une condition) et les *bodies* (des actions), qui sont aussi exprimés en IF.

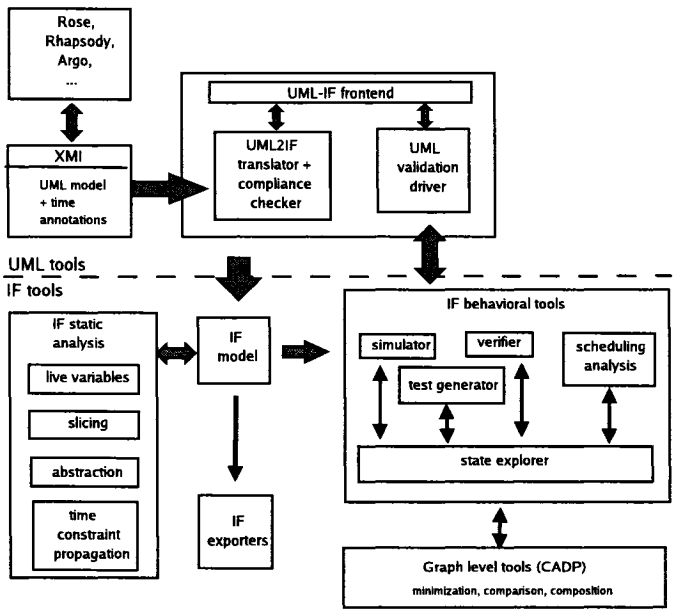

Figure 2.2 Outil IFx

Le modèle sous cette forme (spécification IF) peut être réduit en appliquant l'analyse statique par *slicing* ou d'autres méthodes d'optimisation. Le *slicing* consiste à appliquer un critère d'élimination des parties du programme qui n'influencent pas la formule à vérifier. Cette stratégie de réduction dépend donc directement de l'analyse formelle de la propriété à vérifier.

Pour exprimer les propriétés, l'outil IF utilise des *observateurs*. Ce sont des éléments de modélisation spéciaux exprimés en IF permettant la vérification de propriétés de sûreté comme l'absence de blocage. Pour la vérification, il fait appel à d'autres outils comme EVALUATOR, le model-checker de l'outil CADP.

2.2.2 vUML

Dans (Lilius and Paltor, 1999), les auteurs présentent vUML, un outil automatique de vérification de modèles UML. L'outil permet la modélisation de systèmes concurrents et distribués qui contiennent des objets actifs (objets qui possèdent un thread) et la communication synchrone et asynchrone entre les objets. La modélisation dynamique du système est basée sur les *statecharts* de Harel (Harel, 1987) dont la sémantique a été largement restreinte (les seules actions possibles sont les événements synchronisés) (Porres, 2001).

Les diagrammes UML utilisés pour modéliser le système sont le diagramme de classes, le diagramme de collaboration, le diagramme d'états-transitions (*statechart*) et le digramme de séquences pour exprimer le contre-exemple.

Pour exprimer les propriétés de vivacité et de sûreté, deux stéréotypes, << *progress* >> et << *invalid* >>, sont employées directement sur les états du diagramme d'états-transitions. À l'aide de vUML, les stéréotypes sont traduits directement dans la logique de LTL.

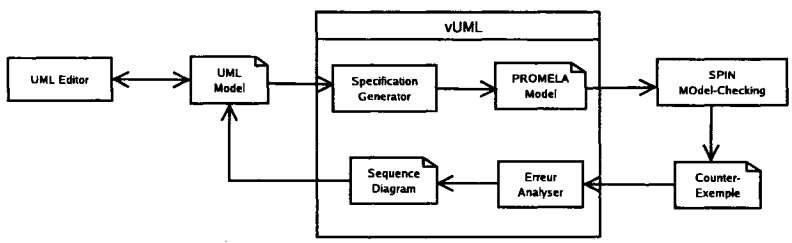

Figure 2.3 Outil vUML

La figure 2.3 illustre une session normale avec vUML. Le concepteur utilise un éditeur d'UML (argoUML, Rational Rose) pour modéliser son système puis invoque vUML pour le vérifier. L'outil fait une traduction dans le langage de spécification PROMELA et vérifie avec SPIN. Si la propriété n'est pas vérifiée, le contre-exemple

fournit par SPIN est traduit par vUML dans un diagramme de séquences. Le concepteur analyse le diagramme et corrige les erreurs. Le processus peut se répéter jusqu'au moment où le modèle est correct.

2.2.3 HUGO

Dans (Knapp and Merz, 2002; Knapp et al., 2002), Knapp et al. présentent l'outil HUGO qui permet, comme vUML, la modélisation en UML par le diagramme de classes et les statecharts d'Harel, et fait appel à SPIN pour la vérification. La différence entre vUML et HUGO tient au fait que les statecharts employés par HUGO comportent moins de limitations et la spécification des propriétés est faite via les diagrammes de collaboration, qui sont plus expressifs que les stéréotypes de vUML. Un diagramme de collaboration décrit le comportement interactif entre les objets et les événements qui se changent, conformément à leurs machines à états.

Certaines restrictions sont tout de même imposées: les événements (les méthodes) n'acceptent pas de paramètres, on ne peut instancier qu'une seule fois une classe et certains constructeurs d'états (dans le statechart) ne sont pas permis comme par exemple les états synchronisés ou les états choix.

Chaque diagramme d'états-transitions est traduit en PROMELA, et le diagramme de collaboration en automate de Büchi. SPIN est invoqué pour la vérification.

2.2.4 JACK

JACK - Just Another Concurrency Kit (Bouali et al., 1994) est un environnement basé sur les algèbres de processus, les automates et les logiques temporelles. Il supporte plusieurs phases de la conception d'un logiciel: la formalisation des be-

soins, la réécriture technique, la simulation du graphe d'exécution, la réduction du graphe et la vérification de propriétés. JACK (figure 2.4) intègre différents outils de spécification et vérification développés dans plusieurs laboratoires (I.E.I.-C.N.R. en Italie et INRIA en France):

1. FC2 (le format commun utilisé en JACK) permet l'échange d'information entre les différents outils intégrés à l'environnement JACK.

2. MAUTO et AUTOGRAPH permettent la spécification textuelle ou graphique du système.

3. MAUTO et HOGGAR permettent aussi par la technique de bisimulation la réduction de communication entre les automates.

4. AMC (Action Model-checker) est le model-checker employé. La logique utilisée est ACTL (action based temporal logic), une logique arborescente qui permet d'exprimer les propriétés sur des systèmes réactifs, caractérisés par des actions. La sémantique est donnée par un système de transitions étiquetées.

Gnesi et al. ont présenté dans (Gnesi et al., 1999) AMC le model-checking `UML` intégré dans l'environnement JACK. Le comportement dynamique du système est modélisé par des diagrammes d'états-transitions qui seront traduits dans un système de transitions étiquetées par une sémantique opérationnelle. La priorité sur les transitions est prise en compte pour réduire les entrelacements et les états hiérarchiques sont permis.

Dans (Gnesi and Mazzati, 2004), les auteurs ont proposé UMC (`UML` on-the-fly Model-Checking) pour vérifier le comportement dynamique des diagrammes d'états -transitions de modèles `UML`, vu comme un ensemble de machines à états communicants. La logique adoptée, $\mu-$ACTL+, est une extension de la logique ACTL, de l'environnement JACK. Elle a la puissance d'expressivité du $\mu-$calcul. Cependant,

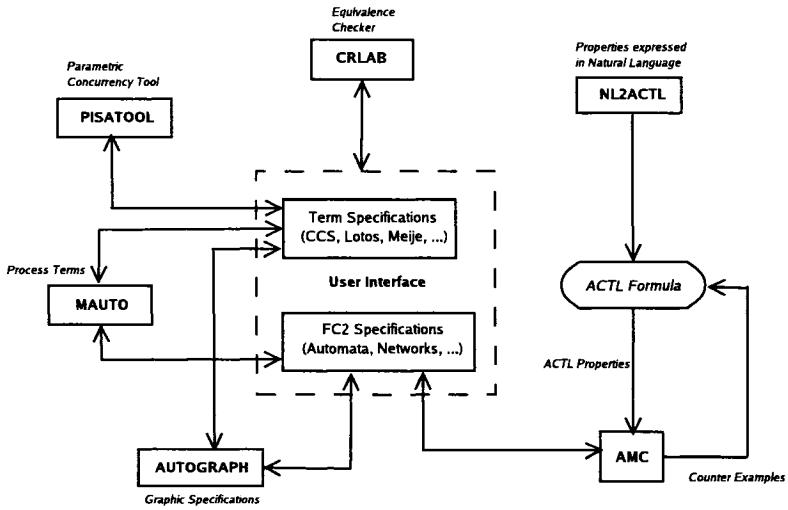

Figure 2.4 Projet JACK

plusieurs suppositions et limitations ont été imposées sur la sémantique des diagrammes d'états UML, par exemple les seuls événements possibles sont des signaux asynchrones (pas d'appel ou retour de méthodes). L'expressivité du μ-calcul est la force de cet outil par rapport à SPIN qui utilise LTL et CADP Evaluator qui utilise le μ-calcul sans alternance. Le papier présente aussi une étude de cas permettant la validation de l'outil.

L'avantage de cet environnement tient du fait que les propriétés à vérifier soient exprimées dans un langage près du langage naturel et traduites ultérieurement par NL2ACTL dans la logique du model-checker.

2.3 Outils - UML avec OCL

2.3.1 USE

Dans (Gogolla et al., 2003), les auteurs décrivent USE (A UML -based Specification Environment), un des premiers outil qui intègre OCL comme langage de spécification de contraintes dans les modèles UML. La syntaxe et la sémantique d'OCL employées en USE sont détaillées dans (Richters and Gogolla, 1998; Richters and Gogolla, 2001).

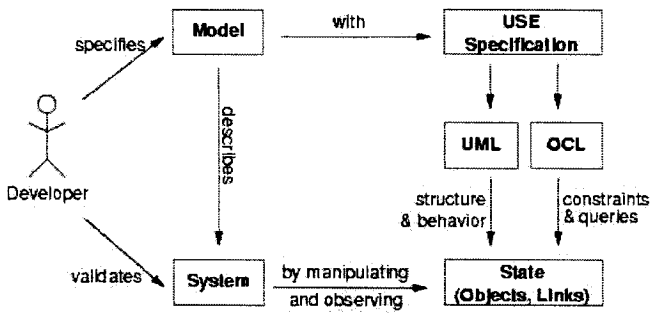

Figure 2.5 Outil USE

La figure 2.5 représente l'environnement USE qui permet la validation d'UML et OCL par la construction de *snapshots* représentant l'état du système à un moment donné. Un *snapshot* est représenté par un diagramme d'objets et il contient les objets, leurs attributs et les liens entre les objets. Une vérification statique est possible en utilisant des invariants.

À l'aide du diagrammes de séquences, l'outil permet la création de tests pour le comportement dynamique. L'utilisateur simule un scénario possible, en appelant ou en retournant des méthodes qui permettent l'interaction entre les objets. Le diagramme de séquences correspondant à ce scénario est construit automatique-

ment. Certaines propriétés du comportement dynamique peuvent être vérifiées par l'utilisation de pré/post conditions.

USE possède une interface agréable et facile à utiliser, mais sa vérification se limite au test. Donc il ne s'agit pas d'une vérification exhaustive, l'outil ne disposant que d'une sémantique statique.

2.3.2 OCLE

Object Constraint Language Environment (OCLE) (Chiorean et al., 2001; Chiorean et al., 2004) est un outil qui possède une sémantique statique pour OCL approfondie au niveau des modèles et métamodèles UML. L'outil utilise une version complète d'OCL, et fait une vérification à travers des règles définies au niveau du métamodèle.

Le fragment d'UML utilisé est composé du diagramme de classes, du diagramme d'objets et du diagramme de cas d'utilisation. Comme dans le cas de l'outil USE, le comportement dynamique est simulé par l'utilisateur en spécifiant les objets dans un état du système, ce qui en limite la vérification dynamique. L'outil permet la génération de code source en Java à partir d'UML.

Quoiqu'OCLE puisse prendre comme entrée pour la vérification les modèles construits avec son propre éditeur UML, ou des modèles fournis par l'outil USE, la puissance de cet outil demeure dans l'analyse statique de modèles et métamodèles et la génération de code.

2.4 Logiques

La spécification OCL ne permet d'exprimer que les propriétés statiques. Exprimer des propriétés dans le temps implique l'utilisation d'une logique temporelle. Les

logiques les plus employées (LTL, CTL, CTL^* ou le μ-calcul), n'intègrent pas OCL. Il faut trouver une fusion entre les deux. Il s'agit donc d'intégrer OCL aux logiques temporelles existantes ou d'étendre OCL par des opérateurs temporaux. Il est à noter de plus que la syntaxe d'OCL n'est pas suffisamment riche pour exprimer certains mécanismes orientés-objets (exemples: quelle méthode est active, quel est l'objet qui exécute une méthode, etc), d'où la nécessité d'intégrer des termes orientés-objet pour pouvoir exprimer un ensemble plus étendu de propriétés sur les modèles UML.

Dans cette section nous présentons trois logiques. La première n'intègre pas OCL. C'est une extension d'une logique classique ($\mu-$calcul modal) utilisée par EVALUATOR (sous-section 2.1.3). La deuxième est un exemple d'une extension OCL et la dernière est une logique temporelle orientée-objet.

2.4.1 Regular alternation-free μ-calculus ($\mathbf{RAF}_\mu$)

RAF_μ est une extension du μ-calcul modal (Kozen, 1983) sans alternance, avec des actions comme dans ACTL (Nicola and Vaandrager, 1990) et avec des expressions régulières comme dans PDL (Fischer and Ladner, Apr). Le μ-calcul modal sans alternance signifie qu'on a au plus une variable libre dans les points fixes et donc on ne peut pas emboîter les opérateurs de points fixes. ACTL est une logique arborescente dans laquelle les opérateurs sont basés sur des actions.

La sémantique de cette logique est définie sur un système de transitions étiquetées: $L = (S, A, T, s_0)$ où S est l'ensemble fini d'états, A un ensemble fini d'actions, $T \subseteq S \times A \times S$ est la relation de transition et $s_0 \in S$ est l'état initial du système. Une transition $(s, a, s') \in T$, indique que le système peut passer de l'état s à l'état s' en effectuant l'action a.

Tableau 2.1 Exemples de propriétés

Catégorie	Propriété	Formule
Sûreté	Absence des actions `Error`	$[\mathtt{T}^*.\mathtt{Error}]\mathsf{F}$
Sûreté	Impossibilité d'avoir une action `Recv` avant une action `Send`	$[\neg\mathtt{Send}^*.\mathtt{Recv}]\mathsf{F}$
Vivacité	Absence de blocage	$[\mathtt{T}^*]\langle\mathtt{T}\rangle\ \mathsf{T}$

Une formule RAF_μ est construite inductivement comme suit à partir de trois types de formules:

$$\begin{aligned}
\texttt{Action formulas}\quad & \alpha ::= a \mid \neg\alpha \mid \alpha_1 \wedge \alpha_2 \\
\texttt{Regular formula}\quad & \beta ::= \alpha \mid \beta_1.\beta_2 \mid \beta_1|\beta_2 \mid \beta^* \\
\texttt{State formula}\quad & \varphi ::= \mathsf{F} \mid \mathsf{T} \mid \varphi_1 \wedge \varphi_1 \mid \langle\beta\rangle\varphi \mid [\beta]\varphi \mid Y \mid \mu Y.\varphi \mid \nu Y.\varphi
\end{aligned}$$

Les formules `action` sont construites en utilisant les opérateurs booléens simples $(\neg, \wedge)$ sur les actions $a \in A$. Les formules `regular` ont pour base les formules `actions` (α) et utilisent comme opérateurs la concaténation (.), le choix (|) et la fermeture transitive-réflexive (*) des expressions régulières. Dans le troisième niveau (`state formula`) on retrouve les formules de μ-calcul modal enrichies par la présence des formules de premier et deuxième niveau, dans les opérateurs modaux $(\langle\beta\rangle, [\beta])$. Les valeurs booléennes T (**True**) et F (**False**) sont les formules atomiques. Y est une variable qui intervient dans le plus petit et le plus grand point fixe: μ, ν. La séquence vide est définie comme $\varepsilon = \mathsf{F}^*$ et l'opérateur de la fermeture transitive est $\beta^+ = \beta.\beta^*$.

Dans le tableau 2.1, nous avons quelques exemples de propriétés exprimées à l'aide de cette logique.

2.4.2 Observation μ-calcul d'OCL - $\mathcal{O}_\mu(OCL)$

Dans (Julian C. Bradfield, 2002) il est proposé une extension d'`OCL` ($\mathcal{O}_\mu(OCL)$) avec des constructeurs temporaux, dans le but d'exprimer *des contrats* plus riches. Un contrat permet d'exprimer, d'une manière explicite et claire, les dépendances entre différentes parties du système. Les contrats les plus communs sont l'`invariant` et la paire `pré/post condition`, mais il est facile d'exprimer d'autres contrats comme la paire `after/eventually` ou n'importe quelle condition de vivacité. Cela facilite la vérification pour les utilisateurs d'`UML` et d'`OCL` qui ne sont pas familiarisés avec l'utilisation des logiques temporelles classiques.

La logique utilisée est basée sur l'*Observational μ-calculus* ($\mathcal{O}_\mu$) (Bradfield et al., 1999; Bradfield and Problem, 1998), une logique temporelle à deux niveaux. Dans le haut niveau, on retrouve le μ-calcul modal (Kozen, 1983) et dans le bas niveau une logique de spécification dite de premier ordre. Si cette logique de spécification est `OCL`, il s'agit de $\mathcal{O}_\mu(OCL)$. La sémantique de $\mathcal{O}_\mu(OCL)$ est formulée sur un système de transitions étiquetées.

La syntaxe formelle de $\mathcal{O}_\mu(OCL)$ est donnée inductivement comme suit:

$$\Phi = \psi \mid \mathbf{T} \mid \mathbf{F} \mid X \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \langle l, C, \phi \rangle \mid [l, C, \phi] \mid \nu X.\Phi \mid \mu X.\Phi \quad \text{où}$$

- ψ est une formule de la logique du bas niveau dite "atomique".
- X est une variable libre de points fixes.
- $\mathbf{T}$, $\mathbf{F}$ sont les valeurs True et False booléennes.
- $\vee$ et $\wedge$ sont les opérateurs booléens *or* et *and*.
- $\langle l, C, \phi \rangle$ est l'opérateur existentiel modal.

- $[l, C, \phi]$ est l'opérateur universel modal.
- νX, μX sont: le plus grand et le plus petit point fixe.

Une formule $\mathcal{O}_\mu$, "observe" le système en évaluant le contenu d'un opérateur modal $\langle l, C, \phi \rangle$ ou $[l, C, \phi]$, où:

- l est une expression action: *send* (envoi d'un message), *return* (réponse à un message), *call* (appel/retour d'une méthode, vu comme une transition complète) et τ des actions internes (ex: création d'objet, affectations, etc).
- C ensemble de cellules mutables ("cells"), qui sont des variables dites de premier ordre, dénommées ainsi pour ne pas les confondre avec les variables de points fixes. Les "cells" réalisent la liaison entre le haut niveau et le bas niveau de la logique. Les variables standards d'`OCL` deviennent donc des variables $\mathcal{O}_\mu$.
- ϕ est une contrainte `OCL`.

Informellement la formule $\mathcal{O}_\mu$: $\langle l, C, \phi \rangle \Phi$ est vérifiée dans un état s, s'il existe une transition étiquetée par une expression action l qui satisfait la contrainte ϕ et qui mène dans un état s' dans lequel Φ est vérifiée.

Un état du système contient comme informations les objets et les méthodes actives.

Un contrat est exprimé par une formule $\mathcal{O}_\mu(OCL)$.

Exemple 2.1 Par exemple l'invariant:

context $ClassName$

inv P

est traduit dans la formule $\mathcal{O}_\mu(OCL)$ suivante:

$$\nu Z.(P \wedge [-\star, \varnothing, \mathbf{T}]Z)$$

Par $-\star$ on comprend toutes les expressions action qui ne sont pas exécutées par l'objet courant. Donc, on peut dire que P est vrai dans tous les états, sauf ceux qui exécutent une méthode sur un objet de la classe contexte *ClassName*.

2.4.3 BOTL

Dans (Rensink et al., 2002; Distefano et al., 2000), les auteurs présentent BOTL (Object-Based Temporal Logic), une logique qui permet la spécification de propriétés statiques et dynamiques pour les systèmes orientés-objet. Cette logique est basée sur la logique temporelle linéaire LTL (Pnueli, 1997) et la spécification `OCL` (OCL 1.1, 1997). La sémantique de cette logique est définie sur un système de transitions non étiquetées $\mathcal{M} = (\mathcal{S}, \rightarrow)$, où $\mathcal{S}$ est l'ensemble d'états du système et $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ est la relation de transitions. Un état est composé par les objets actifs et les méthodes actives.

La syntaxe de BOTL est donnée par la grammaire suivante:

$$\begin{aligned}\varepsilon(\in S_{BOTL}) ::= {} & x \mid \varepsilon.a \mid \varepsilon.\texttt{owner} \mid \varepsilon.\texttt{return} \mid \varepsilon\ \texttt{new} \mid \varepsilon\ \texttt{alive} \mid \omega(\varepsilon, \ldots \varepsilon) \\ & \mid \texttt{with}\ x_1 \in \varepsilon\ \texttt{from}\ x_2 := \varepsilon\ \texttt{do}\ x_2 = \varepsilon\end{aligned}$$

$$\varphi(\in T_{BOTL}) ::= \varepsilon \mid \neg\phi \mid \phi \wedge \phi \mid \exists x \in \tau : \phi \mid \texttt{X}\phi \mid \phi\texttt{U}\phi$$

où S_{BOTL} sont les expressions BOTL inspirées par la syntaxe d'expressions `OCL`, auxquelles sont ajouté des opérateurs nouveaux, spécifiques aux systèmes orientés-

objet:

- x est une variable.
- $\varepsilon.a$ représente l'attribut (paramètre formel), pour l'objet (la méthode) ε, respectivement.
- ε.`owner` dénote l'objet qui exécute la méthode ε.
- ε.`return` représente la valeur de retour de la méthode ε.
- ε `new` exprime que l'objet (la méthode) ε est nouveau (nouvelle), dans l'état courant.
- ε `alive` exprime que l'objet ou la méthode ε est actif (active), dans l'état courant.
- $\omega(\varepsilon, \ldots \varepsilon)$ représente les opérateurs: unaires, binaires et booléens.
- `with` $x_1 \in \varepsilon$ `from` $x_2 := \varepsilon$ `do` $x_2 = \varepsilon$ correspond au `iterate` en `OCL`, qui consiste à parcourir une liste, en effectuant une certaine opération et en retournant un résultat.

T_{BOTL} est l'ensemble des formules temporelles BOTL. Ici, les propriétés atomiques sont remplacées par des expressions BOTL booléennes. Les quantificateurs existentiels et universels sont ajoutés: $\exists x \in \tau : \phi$ exprime que pour tout objet/méthode de type τ la formule ϕ est vérifiée.

La sémantique d'expressions BOTL est donnée par la fonction $[\![\_]\!] : S_{BOTL} \rightarrow (\mathcal{S} \times \Theta) \rightarrow Val_{\perp}$ qui prend comme entrée une expression, un état et un environnement (Θ) et fournit l'évaluation de l'expression. L'environnement contient les valeurs des variables qui sont intégrées dans l'expression à évaluer. L'évaluation peut avoir la

valeur non définie $\perp$. La sémantique des formules BOTL est donnée par la relation de satisfaction $\models \subseteq (\mathcal{S} \times \Theta) \times T_{BOTL}$.

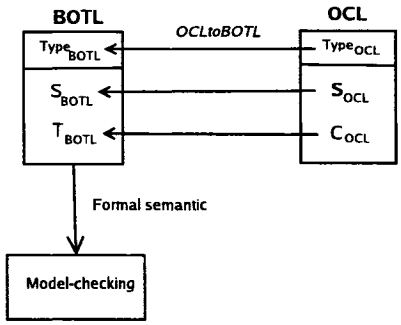

Figure 2.6 Logique BOTL

La syntaxe d'OCL est composée par C_{OCL}, les contraintes OCL (invariant et pré/post condition) et par S_{OCL}, les expressions OCL où l'on retrouve les variables spécifiques à OCL, `self` et `result`, les opérateurs mathématiques simples sur les entiers et booléens, l'opérateur spécial @`pre` et `iterate`.

$$\begin{aligned} \chi(\in C_{OCL}) ::= \ & \texttt{context}\ C\ \texttt{inv}\ e \mid \texttt{context}\ C\ ::\ M(\vec{p})\ \texttt{pre}\ e\ \texttt{post}\ e \\ e(\in S_{OCL}) ::= \ & \texttt{self} \mid x \mid \texttt{result} \mid e\texttt{@pre} \mid e.a \mid \omega(e,\ldots,e) \\ & \mid e.\omega(e,\ldots,e) \mid e \rightarrow \omega(e,\ldots,e) \mid e \rightarrow \texttt{iterate}(x_1; x_2 = e \mid e) \end{aligned}$$

OCL standard (type et syntaxe) est traduit dans la syntaxe de BOTL (figure 2.6). À noter que le seul type de collection supporté par BOTL est le type liste, et donc les ensembles est les multi-ensembles d'OCL sont traduits en conséquence.

Exemple 2.2 L'invariant e, sur le contexte de la classe C est traduit comme suit:

$$\texttt{context } C \texttt{ inv } e \equiv$$
$$\texttt{G}[\forall z \in C \texttt{ ref} : (\neg\exists m_1 \in x.M_1 \texttt{ ref} : \texttt{tt} \wedge \ldots$$
$$\wedge \neg\exists m_k \in x.M_k \texttt{ ref} : \texttt{tt}) \Rightarrow \delta(e)]$$

Par C `ref`, on comprend un objet de la classe C; $m_1.M_1$ `ref` $\ldots m_k.M_k$ `ref` sont des instances de méthodes de la classe C; `tt` est la valeur booléenne Vrai et δ est la fonction qui traduit une expression `OCL` en une expression BOTL.

Informellement, cette traduction veut dire que la condition e doit être vraie dans tous les états qui contiennent des objets actifs de la classe C, mais qui n'exécutent pas de méthodes (de la même classe).

2.5 Synthèse comparatrice des approches

La première partie de l'état de l'art présente une série d'outils qui utilisent comme vérificateur un model-checker exhaustif ou à la volée, pour des systèmes de transitions classiques ou des modèles `UML`. SPIN (sous-section 2.1.2) et CADP (sous-section 2.1.3) sont deux model-checker à la volée mais, qui n'intègrent pas `UML` et donc non plus `OCL`.

Parmi les outils qui utilisent `UML`, il est à noter que, même si une sémantique de comportement dynamique est exprimée, la modélisation est considérablement limitée. Dans vUML (sous-section 2.2.2) et HUGO (sous-section 2.2.3) il y a des restrictions pour les types d'état employés dans les *statecharts*, et les seuls actions acceptées sont les événements synchrones. Il n'y pas de création dynamique

d'objets et l'héritage n'est pas supporté. De plus, le fait qu'ils n'utilisent pas OCL, les paramètres des méthodes ne sont pas pris en compte et l'expressivité des propriétés recherchées est limitée. HUGO nécessite une restriction de plus; les classes ne peuvent être instanciées qu'une seul fois, donc une concurrence très réduite. Pour vérifier de modèle UML à l'aide de Jack (sous-section 2.2.4) ou d'IF (sous-section 2.2.1), une traduction vers leur langage propre est requise. Les modèles UML d'entrée n'intègrent pas OCL et les limitations sont conséquentes: pas des paramètres de méthodes, expressivité limitée des propriétés, etc.

USE (sous-section 2.3.1) et OCLE (sous-section 2.3.2) permettent une modélisation beaucoup plus riche que les autres outils, dû à l'intégration d'OCL. L'héritage est accepté et la création statique d'objets (dirigée par l'utilisateur) est possible. Malheureusement, la vérification des modèles conçus par ces outils se limite au test.

Parmi les outils, ayant à la base une modélisation en UML et OCL, l'outil SOCLe s'avère être le premier à posséder un simulateur exhaustif du comportement dynamique du système, basé sur une sémantique d'un fragment UML et qui intègre les expressions et les contraintes OCL. De plus, il possède deux model-checkers: un, exhaustif, basé sur le μ-calcul et un *à la volée*, basé sur OCL étendue, le sujet de cette thèse.

CHAPITRE 3

STRUCTURE À BASE D'OBJETS ORIENTÉE UML

Ce chapitre commence (section 3.1) par la présentation semi-formelle du fragment UML adopté par SOCLe, en s'appuyant sur les expressions OCL qui précisent la modélisation. La contribution majeure de ce chapitre consiste en la description d'un modèle abstrait orienté-objet, présenté à la section 3.2. Ce modèle nous permettra d'exprimer au chapitre 4 la sémantique des expressions OCL, et la sémantique de la logique adoptée pour le langage de contraintes.

3.1 Fragment UML de Socle

UML est un langage semi-formel, parce que la syntaxe et la sémantique statique (les éléments du système et leurs interconnexions) sont formellement spécifiées, mais la sémantique dynamique est donnée dans le langage naturel. UML au complet est trop complexe pour les besoins et les possibilités d'un environnement de vérification formelle. Par conséquent cela nécessite l'identification d'un sous-ensemble d'UML qui permet toujours de définir raisonnablement le modèle. Ainsi, chaque outil d'aide à la conception utilise un nombre suffisant de diagrammes pour modéliser le système et en propose une sémantique. Plusieurs approches qui s'appuient sur les diagrammes d'états-transitions, ont étés proposées dans la littérature (van der Beeck, 2001; Latella et al., 1999; Wieringa and Broersen, 1998), mais n'intègrent pas OCL. Par constraste, l'outil SOCLe donne la sémantique dynamique de modèles UML pour les diagrammes d'états-transitions (*statechart*), dans lesquels sont incluses les expressions OCL pour décrire les transitions.

Telle qu'elle est présentée dans le rapport technique sur UML d'OMG (Object Management Group) (OMG, 2003b), la sémantique des diagrammes d'états est une extension orientée-objet des statecharts de Harel (Harel and Naamad, 1996). L'outil SOCLe est basé sur ce concept. L'extension orientée-objet est obtenue en sélectionnant le fragment approprié d'actions décrites dans OMG/UML (OMG, 2003b). Certaines restrictions sont imposées par rapport au statechart de Harel (Harel and Naamad, 1996) et le standard (OMG, 2003b): les états hierarchiques ne sont pas supportés et il n'y a pas de notion de priorité pour les transitions. L'aspect statique d'un modèle UML est principalement concentré dans le diagramme de classes. Les interfaces ne sont pas pris en compte dans ce diagramme. Dans l'outil SOCLe, la sémantique est directement inspirée de celle de Java, de Stark et al. (Stärk et al., 2001) (les *threads* sont utilisés pour démarrer les processus). Le diagramme d'objets sert à spécifier la configuration initiale du modèle. La sémantique complète du fragment choisi dans l'outil SOCLe est détaillée dans (Bergeron, 2004).

Nous décrirons la modélisation à l'aide de l'outil SOCLe par le biais d'un exemple qui spécifie un ATM (Automatic Teller Machine). Un ATM est un dispositif électronique qui permet aux clients d'une banque de faire certaines opérations comme des retraits ou des vérifications de solde, sans avoir à consulter un caissier. On se restreint à l'opération de retrait où, informellement, le client s'identifie par son *nip* (numéro d'identification personnel), et où la banque effectue les vérifications nécessaires et fournit l'argent ou une réponse de refus.

3.1.1 Diagramme de classes

Le diagramme de classes (figure 3.1) représente la structure statique du système, en décrivant la structure de chaque classe et leurs associations. Une **classe** est un patron pour créer des instances i.e. des objets. Elle porte un nom et contient un

ensemble d'attributs et un ensemble de méthodes. Les classes qui modélisent notre exemple sont: `ATM`, `Bank`, `BMO` (Banque de Montréal), `BLC` (Banque Laurentienne du Canada).

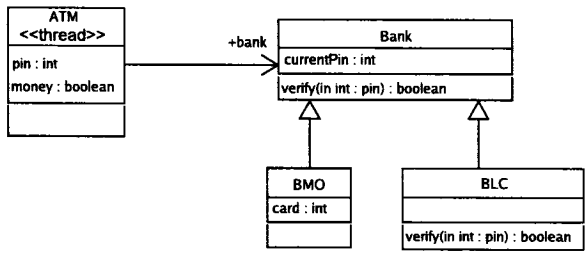

Figure 3.1 Diagramme de classes - ATM

Chaque attribut porte un nom et a un type. Par exemple la classe `ATM` comporte deux attributs: *pin* de type entier et *money* de type booléen.

Une méthode est représentée par sa signature qui contient un nom, des paramètres formels et le type de la valeur de retour. Par exemple *verify(int: pin): boolean* est la seule méthode de la classe `Bank`. Elle demande un seul paramètre formel *pin* de type entier. La valeur de retour est un booléen.

Objet Un objet est une instance d'une classe. Il porte un nom et ses attributs prennent des valeurs concrètes. Le comportement dynamique du système peut changer les valeurs d'attributs d'objets. Un objet exécute une méthode de sa classe. Les objets peuvent être créés ou détruits dynamiquement. Dans la figure 3.3 l'objet `atm1` est une instance de la classe `ATM` et les valeurs de ses attributs sont: *pin = 123* et *money = False.*

La classe `ATM` est un *thread*, cela veut dire que chaque objet de cette classe a une file d'exécution et possède une pile d'appels. Les objets *thread* fonctionnent mutuellement en concurrence.

Navigation Un élément important du diagramme de classes est l'association entre les classes qui permet de traverser les diagrammes de classes et d'objets. Ce processus, qui s'appelle *navigation*, peut être vu comme un chemin qui permet d'aller d'une classe à une autre classe, implicitement, d'un objet à un autre objet, et qui décrit comment un objet d'une classe fait appel à un attribut ou une méthode d'un objet d'une autre classe. Une association porte un nom à son extrémité et elle est considérée comme un attribut de la classe source. Elle est toujours de type liste d'objets de la classe cible. Pour exprimer une navigation, on utilise les expressions `OCL`. La multiplicité est une contrainte restreignant la taille de l'attribut, représentant l'association. L'association *bank* (figure 3.1) lie la classe `ATM` à la classe `Bank`, et elle est considérée comme un attribut de la classe `ATM` dans la navigation.

Héritage Les hiérarchies de classes permettent de gérer la complexité en ordonnant les objets au sein d'arborescences de classes, d'abstraction croissante. L'héritage suppose la *spécialisation* et la *généralisation* et permet la classification des objets. On appelle spécialisation (vision descendante) la possibilité de représenter intégralement tout ce qui a été déjà fait et de l'enrichir, et généralisation (vision ascendante) la possibilité de regrouper dans un seul endroit ce qui est commun à plusieurs (figure 3.2).

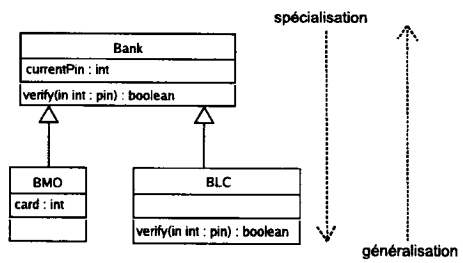

Figure 3.2 Diagramme de classes - Héritage

Le principe de substitution de Liksow (UML, 1997) doit être vérifié: "Il doit être possible de substituer n'importe quelle instance d'une super-classe par n'importe

quelle instance d'une de ses sous-classes, sans que la sémantique d'un programme écrit dans les termes de la super-classe n'en soit affectée." Il permet de déterminer si une relation d'héritage est bien employée pour la classification. Autrement dit, la notion d'héritage agit comme une fonction de sous-typage.

Les classes `BMO` et `BLC` sont des sous-classes de la classe `Bank` (figure 3.2). On dit aussi que `BMO` et `BLC` héritent de `Bank`, cela signifie que `BMO` et `BLC` sont une sorte de `Bank`. Ainsi, là où un objet de type `Bank` peut apparaître, un objet de type `BMO` ou `BLC` peut apparaître aussi. Cela veut dire que si on a à vérifier une propriété dans un contexte de la classe `Bank`, on doit faire la vérification dans tous les objets de la classe `Bank` et dans tous les objets de toutes ses sous-classes.

Une classe dérivée hérite de tous les attributs et des méthodes de la super-classe et peut en ajouter. De plus, une méthode de la super-classe peut être redéfinie dans une sous-classe. Par exemple, la classe `BMO` contient l'attribut *currentPin* de sa super-classe `Bank` et un attribut de plus, *card*, de type entier. Instancier une sous-classe suppose d'instancier ses attributs et ceux de la super-classe (exemple l'objet `bmo1` dans la figure 3.3). La méthode *verify()* est redéfinie dans la sous-classe `BLC`. La redéfinition d'une méthode suppose une même signature (nom, paramètres formels et type de retour).

3.1.2 Diagramme d'objets

En général, le diagramme d'objets modélise un seul état du système. Dans notre cas, le diagramme d'objets représente l'état initial du système. Il contient les objets avec les valeurs de leurs attributs et les associations entre les objets. L'état initial de notre exemple est représenté à la figure 3.3 et contient trois objets: `atm1` instance de la classe `ATM`, `bmo1` et `blc` instances de la classe `BMO` et `BLC` respectivement.

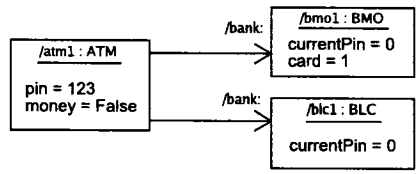

Figure 3.3 Diagramme d'objets

Naviguer dans le diagramme d'objets suppose l'utilisation des associations et des expressions OCL. L'expression OCL *atm1.bank* fournit une liste d'objets qui lie l'objet `atm1` aux autres objets par l'association nommée *bank*. Donc, son évaluation vaut la liste l = [`bmo1,blc1`] (une liste d'objets de type `Bank`) ou bien l'évaluation de l'expression *atm1.bank.currentPin* vaut la liste d'entiers $l_1 = [0, 0]$, qui fait référence à la valeur de l'attribut *currentPin* de chaque élément de la liste d'objets l.

3.1.3 Diagramme d'états-transitions

Les diagrammes d'états-transitions permettent de modéliser le comportement dynamique du système. À chaque classe correspond un diagramme d'états-transitions. Donc, chaque diagramme d'états-transitions modélise le comportement dynamique de la classe. Plus précisément, chaque objet présente une instance du diagramme d'états-transitions, nommée *machine à états*. Le flot de contrôle est spécifié à l'aide des états et des transitions.

3.1.3.1 État

Un état peut être de type initial (`Init`, figure 3.4), de type final (`Fin`), de type simple (`Simple_1`, `Simple_2`) ou de type composé (`Composé`). Un état initial est considéré comme un état simple. Les états simples n'ont pas une signification

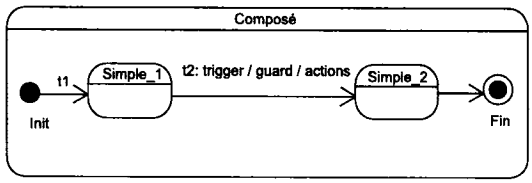

Figure 3.4 Diagramme d'états-transitions (OR)

spécifique. Un état composé peut contenir d'autres états composés ou de formes simples. Un état initial est automatiquement activé quand l'état parent direct est activé. L'atteinte d'un état final implique la désactivation de l'état composé qui l'héberge.

Les états composés supportés sont de type AND et OR. Un état composé de type OR doit contenir au minimum un état initial. Un état composé de type AND ne peut contenir que des états composés de type OR (exemple: `Composé_2` et `Composé_3`, figure 3.5). Un état OR désactivé implique la désactivation de l'état AND qui l'héberge. Les états OR d'un état AND travaillent en concurrence. C'est la deuxième forme de concurrence, supportée par notre outil en plus de multi-threading.

3.1.3.2 Transition

Une transition lie un état source à un état cible. Par exemple pour le diagramme représenté à la figure 3.4 l'état source de la transition *t2* est `Simple_1` et l'état cible est `Simple_2`. Une transition comporte trois parties: *trigger*, *guard* et *actions*. Elle est sensibilisée par le déclencheur (*trigger*). Pour qu'une transition puisse être sensibilisée, il faut aussi que l'état source soit actif. Elle peut être tirée si le garde (*guard*) est vrai et les actions (*actions*) peuvent être effectuées. Une transition

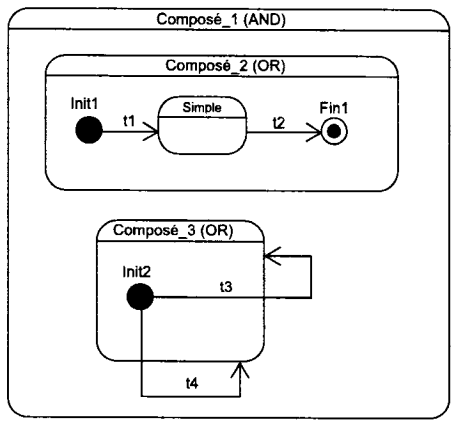

Figure 3.5 Diagramme d'états-transitions (AND)

Tableau 3.1 Syntaxe des déclencheurs

Formel	**Informel**
-	déclencheur vide
$Signal$	réception de signal
$Msig$	appel de méthode
$\%Msig$	retour de méthode
$Msig ::= Meth(Param : Type, \ldots, Param : Type) : Type$	signature de méthode

tirée implique la désactivation de son état source, avec tous les états qu'il compose et l'activation de son état cible. Si l'état cible est composé, tous les états initiaux inclus sont activés.

Les **déclencheurs** possibles sont présentés dans le tableau 3.1. Un déclencheur vide est toujours satisfait. Les signaux sont des événements globaux qui sont désactivés après chaque pas de l'évolution du modèle. Un déclencheur de méthode est satisfait si cette méthode est en cours d'exécution. Un déclencheur de retour de méthode est satisfait si cette méthode est la dernière à avoir terminé son exécution.

À la figure 3.6, on retrouve le diagramme d'états-transitions qui modélise le com-

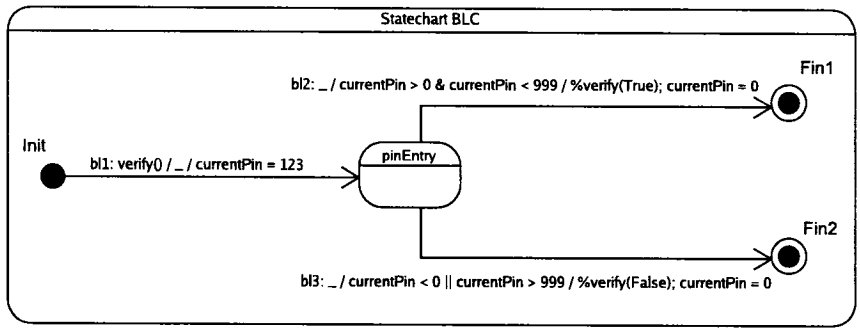

Figure 3.6 Diagramme d'états-transitions - BLC

portement dynamique de la classe BCL. Le déclencheur de la transition *bl1* est un appel de méthode: *verify(pin:int):boolean.* Les deux autres transitions *bl2* et *bl3* ont le déclencheur vide (_).

Un **garde** est une expression OCL booléenne. La syntaxe d'une telle expression peut contenir des constantes, des variables, des attributs d'objets, des opérateurs n-aire et des itérations sur les liste (voir chapitre 4, section 4.3). L'expression OCL: $currentPin > 0$ & $currentPin < 999$, représente le garde de la transition *bl2* (figure 3.6).

Une transition contient une liste d'**actions**. Les actions possibles sont énumérées dans le tableau 3.2. Quand une transition est tirée, la liste d'actions est itérée et les actions sont effectuées. Une action *signal* active le signal courant dans la nouvelle configuration. L'envoi d'un signal peut être effectué à partir de n'importe quelle configuration. Les actions *affectations* mettent à jour les valeurs des attributs d'objets. La *création* d'un nouvel objet, ajoute l'objet dans la liste des objets (actifs). Une nouvelle machine à états lui est associée. Si un objet est *détruit*, il est effacé de la liste d'objets (actifs) et sa machine à états disparaît. Une action *appel de méthode* ajoute la méthode dans la file d'exécution et le *retour d'une méthode* efface

Tableau 3.2 Syntaxe des actions

Formelle	Informelle
Signal	envoi de signal
new Obj	création d'objet
delete Obj	destruction d'objet
Field = OclExp	affectation
OclExp . Meth (OclExp...OclExp)	appel de méthode
super.Meth (OclExp...OclExp)	appel de méthode de la super-classe
%MSig [OclExp]	retour de méthode

cette dernière de la file d'exécution. Toutes les actions sauf *Signal* correspondent intuitivement aux instructions d'une méthode. Elles peuvent seulement être tirées lorsque l'objet qui les possède exécute une méthode.

La transition *bl1* (figure 3.6) a comme action une affectation *currentPin = pin* où *pin* est une expression OCL, constante, et correspond à la valeur du paramètre formel envoyé par l'objet appelant (exemple $atm1$ et $pin = 123$).

Sur la transition *bl2*, nous avons deux actions: une affectation *currentPin = 0*, où 0 est une expression OCL (une constante) et un retour de méthode $\%verify(pin : int)boolean[True]$, où $True$ est aussi une expression OCL booléenne.

Remarque 3.1 Une transition, avec l'état source actif et le déclencheur vide est toujours sensibilisée. Une transition sensibilisée, avec le garde vrai et sans action peut être tirée.

3.1.4 Conclusion sur la modélisation

OCL est partie intégrante d'UML car il participe au mécanisme de navigation dans les diagrammes de classes et d'objets, et permet d'exprimer la dynamique du modèle

dans les diagrammes d'états-transitions. Pour tirer les transitions, il faut avoir la possibilité d'évaluer les gardes et d'effectuer les actions. Une sémantique formelle pour les expressions OCL est donc requise. La deuxième raison qui nécessite l'usage d'une telle sémantique est la vérification des contraintes. Exprimer la sémantique des expressions OCL et celle de la logique de contraintes exige un modèle formel.

3.2 Modèle formel proposé

Afin d'éliminer certaines ambiguïtés qui peuvent intervenir entre la phase de modélisation et l'implantation, et afin d'avoir la possibilité d'appliquer des méthodes formelles de vérification dans la phase de conception, nous développons un modèle formel. Dans cette section, on ne se soucie pas de la façon dont le modèle est généré. Il s'agit d'un système de transitions étendu, une abstraction d'ASM, pour prendre en compte les éléments orientés-objet d'UML (sous-section 3.2.3). Celui-ci permettra au chapitre 4 de donner la sémantique des expressions OCL, et de son extension temporelle. Ainsi, la formalisation implique plusieurs étapes à suivre, à commencer par l'établissement d'un ensemble de types et des valeurs (sous-section 3.2.1). Par la suite, les composantes du modèle seront exposées (sous-section 3.2.2).

3.2.1 Types et valeurs

Pour représenter un modèle UML, on utilise plusieurs ensembles de noms:

- *NVar* ensemble de noms de variables
- *NAtt* ensemble de noms d'attributs.
- *NPar* ensemble de noms de paramètres formels de méthodes.

- *NMet* ensemble de noms de méthodes.
- *NClass* ensemble de noms de classes.

Nous définissons un domaine Type inductivement comme suit:

$$\tau(\in \mathsf{Type}) ::= \mathsf{Void} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{L}(\tau) \mid \mathsf{Obj}^C \mid \mathsf{Met}^C$$

où

- Void ne contient que la valeur triviale ().
- Int est le type entier.
- Bool est le type booléen.
- $\mathsf{L}(\tau)$ dénote le type liste d'éléments de type τ.
  L'élément [] dénote la liste vide et `head::tail` est une liste avec la tête `head` de type τ et le corps `tail` de type $\mathsf{L}(\tau)$.
- Obj^C dénote le type objet de la classe C. Les objets de ses sous-classes sont considérés du même type.
- Met^C dénote le type méthode qui appartient à la classe C ainsi que la méthode qui est redéfinie dans une sous-classe.

L'ensemble de valeurs d'un type $\tau \in \mathsf{Type}$ est dénoté par Val^τ qui se définie comme suit:

$$
\begin{aligned}
Val^{\text{Void}} &= \{()\} \\
Val^{\text{Int}} &= \{\ \ldots -2, -1, 0, 1, 2, \ldots\} \\
Val^{\text{Bool}} &= \{\mathsf{True}, \mathsf{False}\} \\
Val^{\text{L}(\tau)} &= \{[]\} \cup \{\ \texttt{head} :: \texttt{tail} |\ \texttt{head} \in Val^{\tau},\ \texttt{tail} \in Val^{\text{L}(\tau)}\} \\
Val^{\text{Obj}^C} &= \{\texttt{null}\} \cup OId^C \\
Val^{\text{Met}^C} &= MId^{C,M}
\end{aligned}
$$

où

- OId^C représente l'ensemble des instances de la classe C, ainsi que les instances de ses sous-classes.
- $MId^{C,M}$ représente l'ensemble des instances de la méthode M qui appartient à la classe C, ainsi que des instances de la même méthode redéfinie dans ses sous-classes.

L'ensemble de toutes les valeurs est dénoté par Val :

$$Val = Val^{\text{Void}} \cup Val^{\text{Int}} \cup Val^{\text{Bool}} \cup Val^{\text{L}(\tau)} \cup Val^{\text{Obj}^C} \cup Val^{\text{Met}^C}$$

On suppose qu'il existe un élément spécial $\perp \notin Val$ dénotant une valeur non définie. On écrit:

$$Val_{\perp} = Val \cup \{\perp\}$$

Toutes les opérations seront étendues en conséquence. Par exemple, pour l'opération de concaténation sur une liste: $\texttt{head} :: \perp = \perp$ et $\perp :: \texttt{tail} = \perp$. L'ensemble des opérateurs sera présenté dans la chapitre 4, section 4.2.

Note 3.1 On dénote par $\tau(x) \in$ Type le type de x en UML.

3.2.2 Composantes du modèle

Cette étape de la formalisation se concentre sur les déclarations mathématiques qui définissent les éléments significatifs du modèle. Il s'agit en fait de la structure statique décrite par le diagramme de classes et le diagramme d'objets. Afin d'illustrer ce propos, on reprend l'exemple du guichet automatique (ATM) présenté dans la section précédente (3.1), figure 3.1.

Classe

Nous considérons les fonctions partielles suivantes pour formaliser la notion de classe. À chaque variable, attribut ou paramètre formel, on associe un type. À chaque méthode, on associe un ensemble de déclarations de paramètres et un type correspondant au type de la valeur de retour.

$$\texttt{DeclVar} = NVar \pfun \mathsf{Type}$$

$$\texttt{DeclAtt} = NAtt \pfun \mathsf{Type}$$

$$\texttt{DeclPar} = NPar \pfun \mathsf{Type}$$

$$\texttt{DeclMeth} = NMet \pfun (\texttt{DeclPar} \times \mathsf{Type})$$

Une déclaration de classe consiste en un nom et contient les déclarations d'attributs et les déclarations de méthodes.

$$\texttt{DeclClass} = NClass \pfun (\texttt{DeclAtt} \times \texttt{DeclMeth})$$

Soit $D \in \texttt{DeclClass}$ l'ensemble des classes qui composent notre système. Pour toute classe $C \in dom(D)$, nous utilisons les notations suivantes: $C.attrs$ ($\in$ `DeclAtt`) les déclarations des attributs pour la classe C et $C.meths$ ($\in$ `DeclMeth`) les déclarations des méthodes pour la même classe. Donc, formellement une classe est donnée sous la forme suivante: $C = (C.attrs, C.meths)$. S'il est clair que la classe C est le contexte d'une méthode M, alors on utilise $M.pars$ ($\in DeclPar$) pour les paramètres formels de la méthode, et $M.return$ ($\in Type$) pour le type de

Tableau 3.3 Éléments de la classe Bank

Syntaxe dans UML	Sémantique dans la structure
`currentPin: int`	$currentPin \in NAtt$ $\tau(currentPin) = \mathsf{Int}$
`verify(pin:int):boolean`	$verify \in NMet$ $pin \in NPar$ $\tau(pin) = \mathsf{Int}$ $\tau(verify.return) = \mathsf{Bool}$

retour de la méthode. Donc, $C.meths(M) = (M.pars, M.return)$.

Exemple 3.1 Le tableau 3.3 présente les éléments de la classe Bank d'une manière formelle.

L'héritage est formalisé comme une fonction de sous-typage. Nous dénotons par:

- $\preceq_h \subseteq NClass \times NClass$ la relation d'héritage.
- $\preceq_o \subseteq NMet \times NMet$ la relation de redéfinition de méthode (overriding).
- $C'.attrs = \{(C, f) \mid C' \preceq_h C \wedge f \in C.attrs\}$ les attributs de la classe C' qui hérite de C.

Exemple 3.2 Pour l'exemple dans la figure 3.1, formellement on écrit:

$$\texttt{BMO} \preceq_h \texttt{Bank} \ , \ \texttt{BLC} \preceq_h \texttt{Bank}$$

$$\mathit{verify}^{\texttt{BMO}} \preceq_o \mathit{verify}^{\texttt{Bank}}$$

$$\texttt{BMO}.attrs = \{pin : Int, card : Int\}$$

Objet Formellement, un objet est représenté par le nom de la classe et le numéro d'instance de cette classe. Un objet d'une sous-classe fait partie du même ensemble. Soit $C, C' \in NClass$. On définit l'ensemble d'objets de la classe C:

$$OId^C = \{C' \mid C' \preceq_h C\} \times \mathbb{N}$$

Donc, $(C, i) \in OId^C$ représente l'instance i de la classe C.

L'ensemble de tous les objets de toutes les classes:

$$OId = \bigcup_C OId^C$$

Exemple 3.3 (BMO, 1), (Bank,2), (BCL, 3), (BMO, 10)... sont des instances de la classe Bank.

Instance de méthode Les actions (appels, retours) des méthodes permettent de modéliser le comportement dynamique du système. Une méthode appartenant à une classe peut être appelée par un objet d'une autre classe, mais ne peut être exécutée que par un objet de sa classe. Formellement, une méthode est représentée par l'objet qui exécute la méthode, le nom de la méthode et un identificateur qui représente l'instance de la méthode. Soit $M \in NMet$, $C \in NClass$ (tel que $M \in C.meths$) et $o \in OId^C$. On définit l'ensemble des instances de la méthode M, exécutés par des objets de la classe C :

$$MId^{C,M} = OId^C \times \{M \mid M^{C'} \preceq_o M^C\} \times \mathbb{N} \text{ avec } C' \preceq_h C$$

Donc $(o, M, i) \in MId^{C,M}$ représente l'instance i de la méthode M exécuté par l'objet o (instance de la classe C).

L'ensemble de toutes les instances des méthode sera dénoté par MId et il est défini comme suit:

$$MId = \bigcup_C \bigcup_M MId^{C,M}$$

Exemple 3.4 $((\texttt{BMO}, 1), verify, 1)$, $((\texttt{BMO}, 1), verify, 2)$, $((\texttt{BLC}, 2), verify, 1)$ ... sont des instances de la méthode $verify$ (classe Bank). Les deux premières instances sont deux exécutions différentes de la méthode $verify()$ sur le même objet. Cette méthode est héritée par la classe BMO de la super classe Bank. La troisième instance est une exécution d'une méthode redéfinie dans la classe BLC, et exécutée par un objet de cette classe.

3.2.3 Structure à base d'objets orientée UML

On définit maintenant un ensemble minimal de primitives aptes à décrire correctement les mécanismes orientés-objet, en vue de la vérification automatique. Ce modèle est une structure de Kripke $\mathcal{K} =< \mathcal{S}, \mathcal{R} >$, où $\mathcal{S}$ est l'ensemble d'états du système et $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ représente la relation de transitions. Ainsi l'information est concentrée dans les états.

3.2.3.1 État

Une état du système est caractérisé par un ensemble de fonctions partielles, et est représenté par une paire (σ, γ). Soit (σ, γ) l'état courant du système.

$$(1)\ \ \sigma \in \Sigma = OId \rightharpoonup (NAtt \rightharpoonup Val)$$

$$(2)\ \ \gamma \in \Gamma = MId \rightharpoonup ((NPar \rightharpoonup Val) \times Val_{\perp})$$

$$\text{donc, } \mathcal{S} \subseteq \Sigma \times \Gamma$$

où,

(1) *La fonction* σ décrit l'ensemble des objets dans l'état courant. Pour chaque $o \in dom(\sigma)$, $\sigma(o)$ dénote l'environnement de l'objet o dans l'état courant, i.e. les valeurs de ses attributs.

On dénote par $\sigma(o)(a)$ la valeur de l'attribut a de l'objet o.

La fonction σ doit être cohérente avec les déclarations de classes du système, i.e. des classe de D. En ce sens: 1. si $o \in OId^C$, alors $dom(\sigma(o)) = dom(C.attrs)$, 2. si $a \in dom(\sigma(o))$ alors $\sigma(o)(a) \in Val^{C.attrs(a)}$.

(2) *La fonction* γ décrit l'ensemble des instances de méthodes actives dans l'état courant. Pour chaque $m \in dom(\gamma)$, $\gamma(m).env$ dénote l'environnement de la méthode m dans l'état courant, donc les affectations locales de paramètres formels et $\gamma(m).rtn$ la valeur de retour de la méthode. Cette dernière prend une valeur concrète au moment où la méthode a fini son exécution. Dans le cas contraire, cette valeur demeure à $\perp$.

On a donc:

$$\gamma(m) = (\gamma(m).env, \gamma(m).rtn)$$

La fonction γ doit aussi être cohérente sur D. Si $m \in MId^{C,M}$ et p est un paramètre de la méthode m, alors: 1. $dom(\gamma(m).env) = dom(M.pars)$, 2. $\gamma(m).env(p) \in Val^{M.pars(p)}$, et 3. $\gamma(m).rtn \in Val_{\perp}^{M.retour}$.

État initial Graphiquement, l'état initial du système est représenté par le diagramme d'objets. Dans l'état initial aucune méthode n'est active. On dénote par $\mathcal{D}_{obj}$ l'ensemble des objets qui composent le diagramme d'objets.

Si $s_0 = (\sigma_0, \gamma_0) \in \mathcal{S}$ est l'état initial du système, alors:

$$dom(\sigma_0) = \{o \in OId \mid o \in \mathcal{D}_{obj}\}$$
$$dom(\gamma_0) = \varnothing$$

Exemple 3.5 La figure 3.7 illustre trois états dans l'évolution d'ATM: l'état initial, l'état suivant et un état quelconque. Cette exemple permet de rendre plus clair la formalisation.

État initial

L'état initial $s_0 = (\sigma_0, \gamma_0)$ est représenté par le diagramme d'objets et il contient donc un ensemble d'objets. Aucune méthode n'est exécutée, l'ensemble de méthode actives est donc vide.

$dom(\sigma_0) =$ {`atm1`, `bmo1`, `blc1`}
$dom(\gamma_0) = \varnothing$

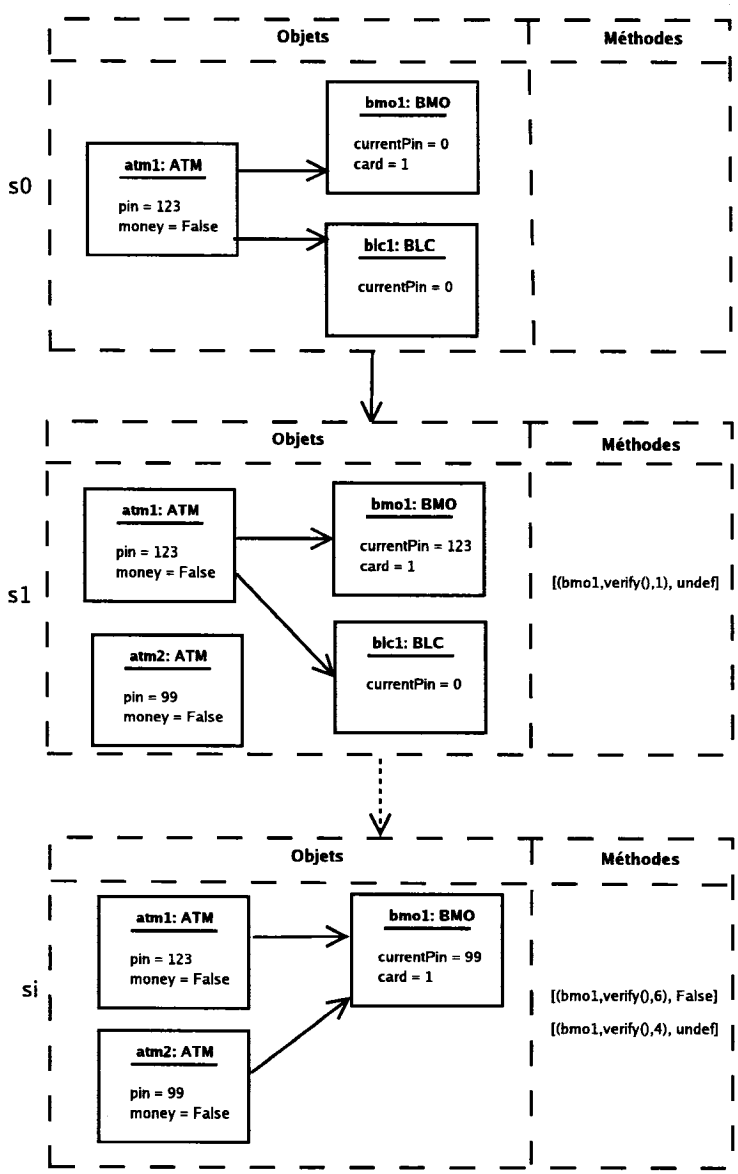

Figure 3.7 Évolution du système

$\sigma_0(\texttt{atm1})(pin) = 123, \quad \sigma_0(\texttt{atm1})(money) = \text{False}$

$\sigma_0(\texttt{bmo1})(currentPin) = 0, \quad \sigma_0(\texttt{bmo1})(card) = 1$

$\sigma_0(\texttt{blc1})(currentPin) = 0$

État 1

Dans l'état $s_1 = (\sigma_1, \gamma_1)$, un nouvel objet s'ajoute à l'ensemble d'objets existants. Une méthode est en cours d'exécution. Il s'agit de la première instance de $verify()$, exécutée par l'objet bmo1, à la demande de l'objet atm1. Comme la méthode n'a pas fini son exécution, la valeur de retour demeure non définie. On observe que la valeur pour l'attribut $currentPin$ de l'objet bmo1 a été changée.

$dom(\sigma_1) = \{\texttt{atm1}, \texttt{bmo1}, \texttt{blc1}, \texttt{atm2}\}$

$dom(\gamma_1) = \{(\texttt{bmo1}, verify, 1)\}$

$\sigma_1(\texttt{atm1})(pin) = 123, \quad \sigma_1(\texttt{atm1})(money) = \text{False}$

$\sigma_1(\texttt{atm2})(pin) = 99, \quad \sigma_1(\texttt{atm2})(money) = \text{False}$

$\sigma_1(\texttt{bmo1})(currentPin) = 123, \quad \sigma_1(\texttt{bmo1})(card) = 1$

$\sigma_1(\texttt{blc1})(currentPin) = 0$

$\gamma_1((\texttt{bmo1}, verify, 1)).env(pin) = 123, \quad \gamma_1((\texttt{bmo1}, verify, 1)).rtn = \perp$

État i

À l'état $s_i = (\sigma_i, \gamma_i)$, l'objet blc1 est détruit et deux instances de la méthode $verify()$ sont dans la pile d'appels. L'instance 4 fini sont exécution.

$dom(\sigma_i) = \{\texttt{atm1}, \texttt{bmo1}, \texttt{atm2}\}$

$dom(\gamma_i) = \{(\texttt{bmo1}, verify, 4), (\texttt{bmo1}, verify, 6)\}$

$\sigma_i(\texttt{atm1})(pin) = 123, \quad \sigma_i(\texttt{atm1})(money) = \text{False}$

$\sigma_i(\texttt{atm2})(pin) = 99, \quad \sigma_i(\texttt{atm2})(money) = \text{False}$

$\sigma_i(\texttt{bmo1})(currentPin) = 99, \quad \sigma_i(\texttt{bmo1})(card) = 1$

$\gamma_i((\texttt{bmo1}, verify, 4)).env(pin) = 99, \quad \gamma_i((\texttt{bmo1}, verify, 4)).rtn = \mathsf{False}$

$\gamma_i((\texttt{bmo1}, verify, 6)).env(pin) = 123, \quad \gamma_i((\texttt{bmo1}, verify, 6)).rtn = \perp$

3.2.3.2 Relation de transitions

La relation de transitions $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ est obtenue en partant de l'état initial et en appliquant la règle de transition d'ASM, expliquée dans l'introduction (chapitre 1, sous-section 1.2.2).

Remarque 3.2 Conformément au standard (OMG, 2002) d'OCL, la sémantique de l'opérateur @**pre** est possible dans l'état de retour de la méthode et fait référence à l'état d'appel.

La relation de transition doit prendre en compte le prédicat qui détecte le retour d'une méthode appelée. Ce prédicat est nécessaire pour assurer la remarque précédente. Si l'instance m d'une méthode est active dans l'état (σ, γ), et devient inactive dans l'état suivant (σ', γ'), alors sa valeur de retour doit être différente de $\perp$. Formellement, on a que si : $[((\sigma, \gamma) \rightarrow (\sigma', \gamma'), m \in dom(\gamma)$ et $m \notin dom(\gamma')) \Rightarrow \exists\ \gamma(m).rtn \neq \perp : \gamma(m) = (\gamma(m).env, \gamma(m).rtn)]$. Il faut comprendre que l'état (σ, γ) ne représente pas forcement l'état où la méthode a été invoquée. Elle aura pu être invoquée bien avant.

3.2.3.3 Fonction historique

Remarque 3.3 Pour faciliter la sémantique de l'opérateur @**pre** dans les expressions OCL, on se propose de garder d'une certaine façon un historique sur les instances de méthodes déjà créées.

Définition 3.1 Nous définissons la fonction dénommé *Fonction historique*:

$$\varepsilon \in \Upsilon = MId \rightharpoonup (\mathcal{S} \cup \{\perp\}) \times 2^{\mathcal{S}}$$

Cette fonction associe à chaque état du système, pour une instance de méthode, une paire telle que le premier élément correspond à l'état où l'instance est créée (un nouvel appel de méthode), et le deuxième élément est un ensemble d'états où la méthode est de retour. On peut avoir plusieurs états de retour pour la même instance de méthode à cause de la concurrence.

Définition 3.2 Par conséquent la structure de Kripke étendue est:

$$\mathcal{K}_\Upsilon =< \mathcal{S} \times \Upsilon, \mathcal{R}_\Upsilon >$$

où $\mathcal{R}_\Upsilon$ est une relation sur $\mathcal{S} \times \Upsilon$ défini comme suit:
$(s, \varepsilon, s', \varepsilon') \in \mathcal{R}_\Upsilon$ ssi $(s, s') \in \mathcal{R}$ et ε' est défini comme suit:

$$\varepsilon'(m) = \begin{cases} (s', \varnothing) & \text{si} \quad m \in dom(\gamma') \text{ et } m \notin dom(\gamma) \\ (s_a, S^r \cup \{s'\}) & \text{si} \quad m \in dom(\gamma') \text{ et } \gamma'(m).rtn \neq \perp \\ \varepsilon(m) & \text{sinon} \end{cases}$$

où $(s_a,\ S^r)$ représente l'image de $\varepsilon(m)$ en s.

Exemple 3.6 La figure 3.8 illustre une partie d'un graphe d'exécution. Deux instances de méthodes sont créées. La mise à jour pour la fonction historique est faite comme suit:

s0 : $dom(\varepsilon) = \varnothing$

s1 : $dom(\varepsilon) = \{m_1\}$

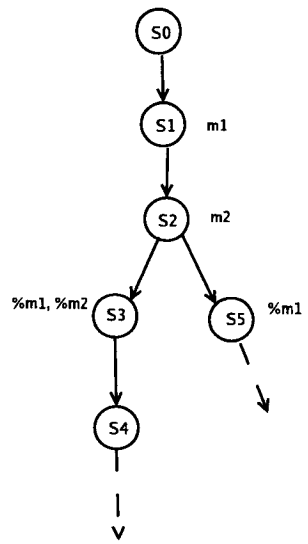

Figure 3.8 Exemple: Fonction historique

$$\varepsilon(m_1) = (s_1, \varnothing)$$

s2 : $dom(\varepsilon) = \{m_1, m_2\}$

$$\varepsilon(m_1) = (s_1, \varnothing)$$

$$\varepsilon(m_2) = (s_2, \varnothing)$$

s3 : $dom(\varepsilon) = \{m_1, m_2\}$

$$\varepsilon(m_1) = (s_1, \{s_3\})$$

$$\varepsilon(m_2) = (s_2, \{s_3\})$$

...

si : $dom(\varepsilon) = \{m_1, m_2, \ldots m_k\}$

$$\varepsilon(m_1) = (s_1, \{s_3, s_5\})$$

$$\varepsilon(m_2) = (s_2, \{s_3\})$$

...

$$\varepsilon(m_k) = (s_a, S^r)$$

CHAPITRE 4

EXTENSION OCL

Ce chapitre présente dans la première partie le langage de spécification OCL et l'extension temporelle que nous proposons. La section 4.1 fait un rappel semi-formel du langage tout en motivant le besoin de sa formalisation et certains restrictions imposées dans l'outil SOCLe. Les sections, 4.2 et 4.3, sont consacrée à la formalisation des expressions OCL, telles qu'elles sont employées dans l'outil SOCLe. La contribution de ce chapitre réside dans la section 4.4, où, nous présentons une extension temporelle d'OCL orientée-objet, capable d'exprimer une classe plus étendue des contrats sur les modèles UML. La section 4.5 exprime les contraintes OCL à l'aide de cette logique. La dernière section, 4.6, présente les algorithmes *à la volée* implantés.

4.1 Vue d'ensemble sur OCL

4.1.1 Motivation de la formalisation d'OCL

L'introduction d'un langage de contraintes en UML représente un pas important vers une formalisation élégante des modèles complexes. OCL est le langage formel de contraintes en UML. OCL permet d'augmenter l'expressivité de modèles UML en ajoutant de l'information supplémentaire sur les diagrammes. Cette information est exprimée sous la forme des expressions et des contraintes OCL.

OCL (OCL 1.1, 1997) est principalement défini d'une manière semi-formelle, en

utilisant le texte descriptif (en language naturel), une syntaxe concrète et plusieurs exemples pour illustrer les significations intentionnelles des expressions. Cette façon de présenter, aussi pédagogique qu'elle soit, est inadéquate pour formaliser une sémantique complète d'OCL. Ceci requiert une définition formelle. Dès plus les contraintes OCL permettent de valider les modèles UML que d'une manière statique. En vue de pouvoir exprimer des propriétés de comportements dynamiques pour des tels modèles, on est censé d'étendre OCL avec des opérateurs temporaux et des notations orientées-objet. Donc la formalisation d'OCL est doublement motivée.

4.1.2 OCL en exemples

Avant de donner la syntaxe et la sémantique des expressions OCL, il faut présenter quelques exemples d'utilisation des expressions OCL sur un modèle UML, pour mieux comprendre l'avantage qu'OCL apporte, et certaines problèmes qui peuvent apparaître.

Les exemples d'expressions OCL qui suivent ont comme support une petite étude de cas modélisée en SOCLe. Cette étude de cas ne représente pas forcément un modèle réel et, donc, ne se soucie pas nécessairement de la bonne forme du modèle. On l'utilise simplement pour justifier l'application des expressions OCL dans les modèles UML. Cet exemple est d'une certaine manière une forme réduite de l'étude de cas qui est présentée en détail dans le chapitre 5.

La structure statique du modèle est représentée par le diagramme de classes (figure 4.1) et l'état initial du système est graphiquement représenté à la figure 4.2. Le comportement dynamique du système n'est présenté que partiellement par un seul diagramme d'états-transitions correspondant à la classe `Archives` (figure 4.3).

Informellement, cet exemple modélise un usager qui essaie d'accéder à des do-

cuments rangés dans des archives. Ces documents sont caractérisés par certaines propriétés de sécurité. Ces archives sont structurées par domaine. Un domaine contient plusieurs documents. Un document est caractérisé par son niveau de sécurité (*level*) et par son cavéat (*caveat* [1]). Un usager est caractérisé par son âge (*age*), le niveau de visibilité (*clearance*) et le pays d'origine (*country*). Les archives vérifient les droits d'accès aux documents par la méthode *access()*. Elle peut aussi recevoir de nouveaux documents par la méthode *checkIn()*.

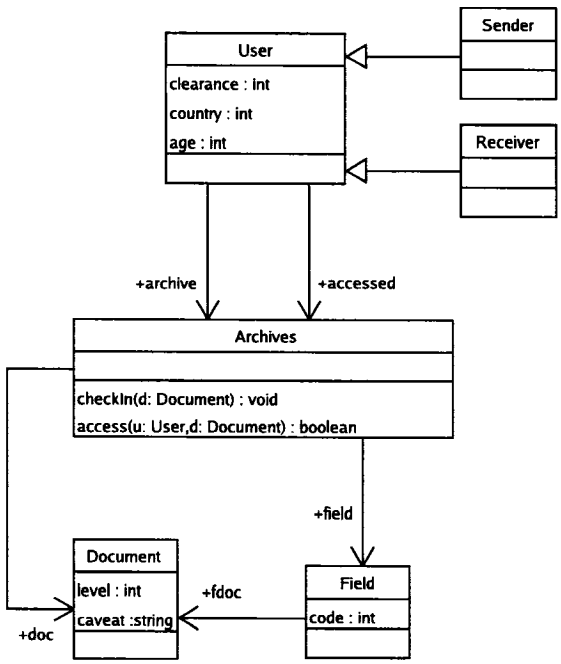

Figure 4.1 Diagramme de classes - Archives

Exemple 4.1 Par une simple contrainte OCL, on peut limiter la valeur d'un attribut.

[1] Un autre critère de classification, pour les documents sécurisés

L'âge de l'usager doit être positif.

$$\textbf{context}: \texttt{User}$$

$$\textbf{exp}: self.age > 0$$

La variable *self* fait référence à l'objet courant du contexte. Ici, il s'agit des objets de la classe `User`. Si le modèle n'a aucun comportement dynamique, la vérification de cette contrainte se réduit à vérifier la condition pour toute instance de la classe `User` dans l'état initial. Dans notre cas, il n'y a qu'un usager, `s` et son âge est 23 (voir la figure 4.2). La contrainte est donc vérifiée. Si par contre, le modèle présente un comportement dynamique, cette contrainte doit être vérifiée dans chaque configuration du système. `OCL` ne permet pas d'exprimer de telle conditions (sur un chemin d'exécution).

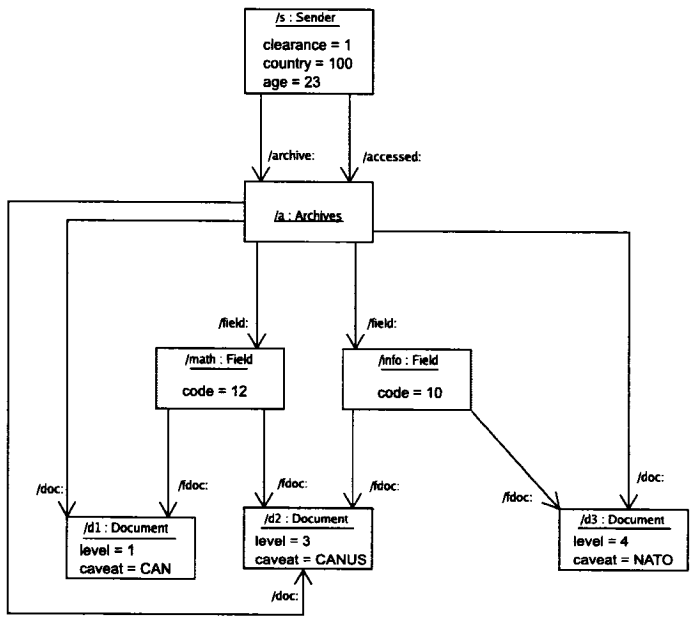

Figure 4.2 Diagramme d'objets - Archives

Exemple 4.2 On peut imposer que le niveaux de sécurité de chaque document des

archives doit être compris entre 0 et 3.

context : `Archives`

exp : $self.doc \rightarrow \textbf{forall}\{d \mid d.level >= 0 \ \& \ d.level < 4\}$

Dans ce cas, toujours pour la modélisation statique (donc, pour l'état initial) la contrainte n'est pas vérifiée car le niveau de sécurité du document `d3` (dans le figure 4.2) ne respecte pas la condition .

Exemple 4.3 On peut valider l'existence d'un document des archives qui a le cavéat CANUS.

context : `Archives`

exp : $self.doc \rightarrow \textbf{exist}\{d \mid d.caveat = \text{CANUS}\}$

La contrainte est vérifiée, le document `d2` a le cavéat demandé.

Les deux exemples suivants mettent en valeur le rôle des expressions `OCL` dans la navigation des diagrammes `UML`. La sémantique informelle de ces exemples met en évidence l'application de l'opération de *flattening* sur les listes imbriquées de même type, et le fait de retrouver le même résultat si on a la possibilité de naviguer de plusieurs façons pour la même requête. Ils utilisent le diagramme d'objets (figure 4.2). Il n'y a qu'un seul usager (`s`) et une seule archive (`a`), qui est structurée en deux domaines (`math`, `info`). Le domaine des mathématiques contient deux documents (`d1` et `d2`) et le domaine de l'informatique deux documents (`d2` et `d3`).

Exemple 4.4 L'expression suivante va chercher tous les documents des archives en

utilisant les associations *field* et *fdoc*.

context : `Archives`

exp : $self.field.fdoc$

evaluation : flat[[d1,d2],[d2,d3]] = [d1,d2,d3]

Exemple 4.5 L'expression suivante va chercher tous les documents des archives en utilisant l'association *doc*.

context : `Archives`

exp : $self.doc$

evaluation: [d1,d2,d3]

Les types en OCL sont présentés dans la section suivante 4.2. Ici nous nous contenterons d'illustrer dans l'exemple qui suit, l'ambiguïté qui peut être causée par l'utilisation des collections telles que les ensembles et les multi-ensembles. Un ensemble est une collection composée par des éléments uniques et non ordonnés. On suppose que le type de collection employé est l'ensemble.

Exemple 4.6 L'expression suivante est une chaîne de caractères obtenue par la concaténation d'attributs *caveat* de chaque document des archives a, dans une variable

cavs.

a: Archive

exp : a.*doc* → **iterate**{*d* : Document ; cavs = " "|cavs.**concat**(*d.caveat*)}

evaluation: **résultat 1:** cav =″ CAN CANUS NATO″

résultat 2: cav =″ CAN NATO CANUS″

résultat 3: cav =″ NATO CANUS CAN″

...

On obtient plusieurs résultats selon l'ordre dans lequel on prend les éléments dans l'ensemble. Dans l'outil SOCLe, on n'emploie que les listes comme type de collection, pour éviter ce genre de problème.

Les deux exemples qui suivent sont reliés au diagramme d'états-transitions qui modélise le comportement dynamique de la classe `Archives` (figure 4.3). Dans ce cas le contexte des expressions `OCL` est implicitement la classe `Archives`.

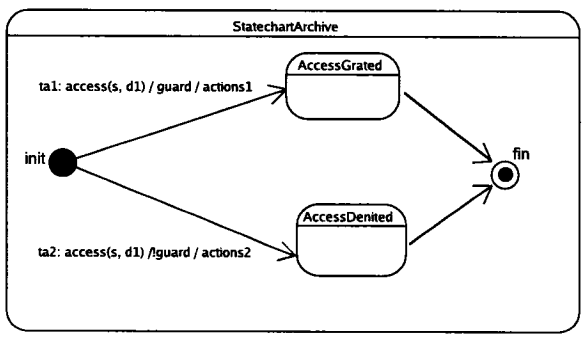

Figure 4.3 Diagramme d'états-transitions - Archives

Exemple 4.7 Un garde sur la transition *ta1* sert à spécifier que le niveau de visibilité

d'un usager doit être plus grand ou égal au niveau de sécurité du document.

$$\textbf{exp}: \ \mathtt{s}.level >= \mathtt{d1}.level$$

Exemple 4.8 Une action de la transition *ta1* est spécifiée par une affectation.

$$\textbf{exp}: \ accessed = accessed :: self.doc.\text{head}$$

Si l'usager reçoit le droit d'accès au document, celui-ci est enregistré dans la variable (*accessed*). Plus précisément ici, on ajoute l'élément de la tête de liste de documents existant dans l'archive.

4.2 Types en OCL

4.2.1 Types OCL standard

Une définition formelle d'OCL présuppose des déclarations de types, une syntaxe et une sémantique. À la figure 4.4, on retrouve les types disponibles en OCL conformément à (OCL 1.1, 1997). Les flèches indiquent les sous-types et les relations entre les types. Tous les types sont directement ou indirectement des sous-types du type abstrait OclAny.

Les types de base sont Real, Integer, Boolean, et String. Le type Integer est un sous-type de Real. On peut dire que le domaine des entiers est inclus dans le domaine des réels ($Val^{Integer} \subset Val^{Real}$).

Enumeration est une extension des types de base. Par exemple, si on a *Couleur*: enum{blanc, rouge, bleu}, alors le domaine de valeurs est donné par l'énumération de la valeur des chaînes de caractères, $Val^{Couleur} = \{\text{blanc}, \text{rouge}, \text{bleu}\}$.

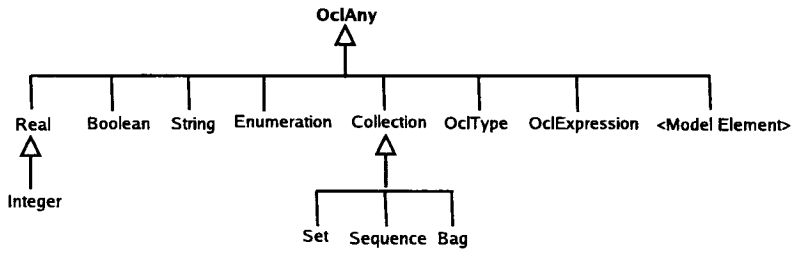

Figure 4.4 Hiérarchie de types en OCL

Les types complexes exprimés par les collections: ensemble, multi-ensemble et liste, sont construits à partir des types simples. Par exemple, Set(Integer) signifie un ensemble d'entiers. Les collections ne peuvent pas être emboîtées. Une collection emboîtée est automatiquement traduite par l'opération de *flattening*, à un ensemble, multi-ensemble ou liste d'un type de base. Par exemple, Bag(Set(Integer)) sera traduit en Bag(Integer).

Les autres types complexes OclType et OclExpression sont construits inductivement en utilisant les types de base et les collections.

4.2.2 Types OCL dans Socle

Types Le domaine de types OCL est défini inductivement comme suit:

$$\tau(\in \mathsf{Type}_{OCL}) ::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Obj}^{C} \mid \mathsf{L}(\tau)$$

Les types simples sont les types des entiers, des booléens et des objets (instance de la classe C). Un seul constructeur pour le type complexe est utilisé: la liste de types simples. L'ensemble de types OCL est un sous ensemble de types UML, et sont donc interprétés de la même façon.

Valeurs Les valeurs correspondantes à ces types sont expliquées en détails dans la section 3.2 du chapitre 3. Le domaine de valeurs est étendu pour chaque type avec la valeur non définie ($\perp$).

$$Val_{OCL} = Val/\{Val^{Void} \cup Val^{Met^C}\}$$
$$Val^{\perp}_{OCL} = Val_{OCL} \cup \{\perp\}$$

Opérations Les opérations possibles sont présentées exhaustivement à la figure 4.5. On retrouve les opérations arithmétiques de base sur les entiers, les booléens et les listes. Chaque opération est étendue à la valeur non définie $\perp$.

Exemple 4.9 Soit x, y deux entiers et a, b deux booléens. L'extension de l'*addition* sur les entiers, et le *ou* logique sur les booléens est défini de la façon suivante:

$$x + y = \begin{cases} x + y & \text{si } x \neq \perp \text{ et } y \neq \perp \\ \perp & \text{sinon} \end{cases}$$

$$a \vee b = \begin{cases} \text{True} & \text{si } a = \text{True ou } b = \text{True} \\ \text{False} & \text{si } a = \text{False et } b = \text{False} \\ \perp & \text{sinon} \end{cases}$$

Variables On dénote par Var_{OCL} l'ensemble des noms de variables employées dans le langage de spécification OCL. Il inclut les variables locales utilisées dans les collections ($NVar$), les paramètres formels de méthodes ($NPar$) et la variable

Val	True	:	Bool	
	False	:	Bool	
	$\ldots, -1, 0, 1, \ldots$	:	Int	
$\omega : Bop$	$+$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int}$	Addition
	$-$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int}$	Soustraction
	$*$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int}$	Multiplication
	$/$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int}$	Division
	$\%$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Int}$	Modulo
	$<$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Bool}$	Plus petit
	$<=$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Bool}$	Plus petit ou égal
	$>$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Bool}$	Plus grand
	$>=$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Bool}$	Plus grand ou égal
	$==$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Bool}$	Égalité
	$!=$	:	$\texttt{Int} \times \texttt{Int} \rightarrow \texttt{Bool}$	Inégalité
	$\&$	:	$\texttt{Bool} \times \texttt{Bool} \rightarrow \texttt{Bool}$	ET
	$\vert$	:	$\texttt{Bool} \times \texttt{Bool} \rightarrow \texttt{Bool}$	OU
	$\vert\vert$	:	$\texttt{Bool} \times \texttt{Bool} \rightarrow \texttt{Bool}$	OU exclusif
	$\Rightarrow$	:	$\texttt{Bool} \times \texttt{Bool} \rightarrow \texttt{Bool}$	Implication Booléen
	@	:	$\texttt{L}^\tau \times \texttt{L}^\tau \rightarrow \texttt{L}^\tau$	Concaténation des listes
	::	:	$\tau \times \texttt{L}^\tau \rightarrow \texttt{L}^\tau$	Construction des listes
$\omega : Uop$	$+$	:	$\texttt{Int} \rightarrow \texttt{Int}$	Addition unaire
	hd	:	$\texttt{L}^\tau \rightarrow \tau$	Tête de la liste
	$tail$	:	$\texttt{L}^\tau \rightarrow \texttt{L}^\tau$	Corps de la liste
	$sort$	:	$\texttt{L}^\tau \rightarrow \texttt{L}^\tau$	Tri d'une liste
	$-$	:	$\texttt{Int} \rightarrow \texttt{Int}$	Soustraction unaire
	$!$	:	$\texttt{Bool} \rightarrow \texttt{Bool}$	Négation

Figure 4.5 Opérateurs ω

spécifique d'`OCL` *self.*

$$Var_{OCL} = NVar \cup NPar \cup \{self\}$$

La déclaration des variables est une fonction partielle. Chaque variable a un nom et un type.

$$\texttt{DeclVar}_{OCL} \quad = \quad Var_{OCL} \rightharpoonup \texttt{Type}_{OCL}$$

4.3 Expressions OCL

Cette section présente la syntaxe et la sémantique d'expressions OCL employées par l'outil SOCLe.

4.3.1 Syntaxe des expressions OCL

Les expressions OCL possèdent une syntaxe formelle. L'ensemble des expressions OCL est défini par la grammaire suivante:

$$e(\in E_{exp}) \quad ::= \quad v \mid x \mid \omega(e_1, \ldots, e_n) \mid e\,.\,a \mid e_1\,.\,\textbf{iterate}\ (x_1\ ;\ x_2 = e_2 \mid e_3\ ) \mid e\ @\textbf{pre}$$

où:

- v est une valeur dans le domaine de valeur, $Val^{\perp}_{OCL}$
- x est une variable dans de domaine de variable, Var_{OCL}
- ω dénote un opérateur générique booléen, arithmétique ou de listes, d'arité n (figure 4.5).
- e.**a** utilise l'opérateur de navigation "**.**". L'expression "**.a**" est connue aussi sous le nom d'*attribut* de navigation, où **a** est un attribut et la sous expression e, une référence à un objet ou une liste de références à des objets. Si e est une liste d'objets, alors e.**a** sera une liste d'attributs.
- e_1.**iterate** $(x_1\ ;\ x_2 = e_2 \mid e_3\ )$ correspond à l'itération sur la liste qui est fournie par l'évaluation de l'expression e_1 lorsqu'on effectue l'opération suivante sur ses éléments: x_2 est initialisée à e_2, la variable x_1 prend successivement la valeur de chaque élément de la liste e_1; à chaque pas, l'expression e_3

est évaluée et affectée à la variable x_2.

Par exemple si on a: $e_1 = [1, 2, 3]$, $e_2 = 0$ et $e_3 = x_1 + x_2$, le résultat de l'itération est 6. On calcule la somme d'éléments de la liste $[1, 2, 3]$.

Les opérateurs prédéfinis à partir de l'itération de liste sont présentées à la figure 4.6.

- **@pre** est un opérateur typique à OCL qui fait référence à la valeur de l'opérande au moment de l'appel de la méthode, et peut être appliqué au moment du retour de la méthode (dans un état post-condition).

`size`	**iterate**(v_1; $v_2 = 0 \mid v_2 + 1$)	Taille d'une collection
`forall`$\{v_1 \mid exp\}$	**iterate**(v_1; $v_2 =$ **True** $\mid v_2$ & exp)	Quant. universel
`exists`$\{v_1 \mid exp\}$	**iterate**(v_1; $v_2 =$ **False** $\mid v_2 \mid exp$)	Quant. existentiel
`unique`$\{v_1 \mid exp\}$	**iterate**(v_1; $v_2 =$ **False** $\mid v_2 \mid\mid exp$)	Unicité
`filter`$\{v_1 \mid exp\}$	**iterate**(v_1; $v_2 = exp_1 \mid v_3 = exp_2; v_2 = v_3$)	Filtrage

Figure 4.6 Opérateurs de collections

4.3.2 Sémantique des expressions OCL

Avant de présenter la sémantique d'OCL, voici quelques définitions préliminaires.

Définition 4.1 Une variable d'environnement est une fonction partielle qui assigne des valeurs aux variables:

$$\Theta = Var_{\text{OCL}} \rightharpoonup Val_{\text{OCL}}$$

Définition 4.2 On définit l'opérateur de typage $\vdash$, une fonction qui assigne un type à une expression OCL. On écrit $\vdash e : \tau$ quand l'expression e est de type τ. Cette

fonction est définie inductivement sur la structure de l'expression à partir des règles de typages qui se retrouvent à la figure 4.7.

On fait appel à un environnement de typage Λ, qui accumule les variable OCL et leurs types: $\Lambda = \{x_1 : \tau_1,\ x_2 : \tau_2,\ self : \tau_3 \ldots\}$. Si la variable est *self*, alors son type est le contexte de l'expression. Si l'expression appartient à un diagramme d'états-transitions alors le contexte est la classe qui est modélisée par ce diagramme. Si l'expression fait partie d'un contrainte, alors le contexte est donné explicitement, par le patron de la contrainte (la classe de l'invariant si l'expression exprime un invariant ou la classe de la méthode dans le cas d'un pré/post condition).

Par la virgule dans la règle (1) on comprend l'ajout d'une nouvelle variable libre de type τ à l'environnement Λ (qui peut être vide au début). Informellement on dit, que le type de x en Λ est τ ($\Lambda \vdash x : \tau$). La règle (2) correspond au cas où l'expression est une constante, et les règles (3) et (4) correspondent aux expressions qui contient les opérateurs unaires respectivement binaires. Par exemple True est une constante de type Bool. Les règles (5), (6) et (7) correspondent aux expressions orientées-objet: le type d'attributs d'un objet ou une liste d'objets. La dernière, (8), détermine le type d'une expression qui contient un *iterate* sur les collections.

Exemple 4.10 On se propose de déterminer le type de l'expression OCL suivante $e(\in E_{exp}) = \texttt{d1}.level > \texttt{d2}.level$ qui réfère à l'exemple de la section précédente 4.1.2. L'arbre de typage et le suivant:

$$\dfrac{\dfrac{\vdash \texttt{d1} : \texttt{Document} \quad \vdash level : \texttt{Int}}{\vdash \texttt{d1}.level : \texttt{Int}}(5) \quad \dfrac{\vdash \texttt{d2} : \texttt{Documet} \quad \vdash level : \texttt{Int}}{\vdash \texttt{d2}.level : \texttt{Int}}(5)}{\vdash (\texttt{d1}.level < \texttt{d1}.level) : \texttt{Bool}}\ (3)$$

En utilisant les règle (5) et (3) on obtient que le type de l'expression est booléen. On écrit: $\vdash e$: Bool pour dire que e est de type Bool.

(1) $\dfrac{\Lambda, x : \tau}{\Lambda \vdash x : \tau}$

(2) $\dfrac{}{\Lambda \vdash const : \tau}$ if $const : \tau$

(3) $\dfrac{\Lambda \vdash exp_1 : \tau_1 \quad \Lambda \vdash exp_2 : \tau_2}{\Lambda \vdash exp_1\ bop\ exp_2 : \tau_3}$ if $bop : \tau_1 \times \tau_2 \rightarrow \tau_3$

(4) $\dfrac{\Lambda \vdash exp_1 : \tau_1}{\Lambda \vdash uop\ exp_1 : \tau_2}$ if $uop : \tau_1 \rightarrow \tau_2$

(5) $\dfrac{\Lambda \vdash exp_1 : C \quad \Lambda \vdash D/f : \tau}{\Lambda \vdash exp_1.D/f : \tau}$ if $C \preceq_h D$ and $(D, f) \in C.attrs$

(6) $\dfrac{\Lambda \vdash exp_1 : \mathrm{L}(C) \quad \Lambda \vdash D/f : \tau}{\Lambda \vdash exp_1.D/f : \mathrm{L}(\tau)}$ if $\tau \neq \mathrm{L}(\tau')$, $C \preceq_h D$ and $(D, f) \in C.attrs$

(7) $\dfrac{\Lambda \vdash exp_1 : \mathrm{L}(C) \quad \Lambda \vdash D/f : \mathrm{L}(\tau)}{\Lambda \vdash exp_1.D/f : \mathrm{L}(\tau)}$ if $C \preceq_h D$ and $(D, f) \in C.attrs$

(8) $\dfrac{\Lambda \vdash exp_1 : \mathrm{L}(\tau_1) \quad \Lambda \vdash exp_2 : \tau_2 \quad \Lambda, var_1 : \tau_1, var_2 : \tau_2 \vdash exp_3 : \tau_3}{\Lambda \vdash exp_1.\mathbf{iterate}(var_1;\ var_2 = exp_2 \mid exp_3) : \tau_3}$

Figure 4.7 Règles de typage d'OCL

Maintenant on dispose de tous les préliminaires nécessaires pour exprimer la sémantique des expressions OCL. Elle est donnée par la fonction:

$$[\![\_]\!] : E_{exp} \rightarrow ((\Sigma \times \Gamma) \times \Upsilon \times \Theta \times MId) \rightarrow Val^{\perp}_{\mathtt{OCL}}$$

Elle est définie par induction sur la structure des expressions OCL.

Soit $e \in E_{exp}$, une expression OCL, $\theta \in \Theta$ une variable d'environnement, $s = (\sigma, \gamma) \in \mathcal{S}$ un état et $\varepsilon \in \Upsilon$ la fonction historique correspondante à l'état s, alors l'évaluation de la fonction $[\![\_]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon}$ est définie comme suit:

- Si l'expression OCL est une valeur de la forme $e \equiv v$:

$$[\![v]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = v$$

- Si l'expression OCL est une variable de la forme $e \equiv x$, on a:

$$[\![x]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = \theta(x)$$

- Dans le cas d'un opérateur ω, i.e. $e \equiv \omega(e_1, e_2, \ldots e_n)$:

$$[\![\omega(e_1,\ e_2, \ldots e_n)]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = [\![\omega]\!]([\![e_1]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon}, [\![e_2]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon}, \ldots [\![e_3]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon})$$

- Dans le cas d'une navigation, i.e. $e \equiv e'.a$, on doit distinguer plusieurs sous-cas selon le type de l'expression e'

si $\vdash e' : \mathsf{Obj}^C$

$$[\![e'.a]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = \sigma([\![e']\!]^{\theta,m}_{\sigma,\gamma,\varepsilon})(a)$$

si $\vdash e' : \mathsf{L}(\mathsf{Obj}^C)$ et $\tau(a) \in \{\ \mathsf{Int}, \mathsf{Bool}, \mathsf{Obj}^C\}$ et $[\![e']\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = [o_1, o_2, \ldots o_n]$

$$[\![e'.a]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = [\sigma(o_1)(a), \sigma(o_2)(a), \ldots \sigma(o_n)(a)]$$

si $\vdash e' : \mathsf{L}(\mathsf{Obj}^C)$ et $\tau(a) \in \mathsf{L}(\tau)$ et $[\![e']\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = [o_1, o_2, \ldots o_n]$

$$[\![e'.a]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = [\sigma(o_1)(a) @ \sigma(o_2)(a)\ \ldots @ \sigma(o_n)(a)]$$

- Dans le cas d'une expression OCL qui contient l'opérateur **@pre**, i.e $e \equiv e_1$**@pre**:

$$[\![e]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = \begin{cases} \bot & \text{si } m \notin dom(\varepsilon) \\ \bot & \text{si } \varepsilon(m) = (s_a, \varnothing) \\ [\![e_1]\!]^{\theta,m}_{\sigma_a,\gamma_a,\varepsilon} & \text{si } \varepsilon(m) = (s_a, S^r) \end{cases}$$

- Si l'expression OCL est de la forme $e \equiv e_1$**.iterate**$\{x_1; x_2 = e_2 \mid e_3\}$, alors $[\![e]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon}$ est défini par le pseudo-code CAML suivant:

let $v_1 = [\![e_1]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon}$ **in**
  if size$(v_1) = 0$ **then**
    []
  else
    let $v_2 = [\![e_2]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon}$ **in**
    let $v_3 =$ **head** (v_1) **in**
    let $v_4 = [\![e_3]\!]^{\theta[x_1 \mapsto v_3, x_2 \mapsto v_2],m}_{\sigma,\gamma,\varepsilon}$ **in**
      if size$(v_1) = 1$ **then**
        v_4
      else
        $[\![$tail(v_1)**.iterate**$\{x_1; x_2 = v_4 \mid e_3\}]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon}$

4.4 Extension temporelle d'OCL - OCL^{EXT}

Pour exprimer les contrats standards définis dans OCL (OCL 1.1, 1997) (l'invariant et les pré/post-conditions, sur le fragment UML de SOCLe) et pouvoir en ajouter des nouveaux, une logique temporelle est requise. Cette logique doit permettre d'exprimer et d'évaluer les expressions OCL qui modélisent le système, et de vérifier

les contraintes sur les chemins d'exécution. On s'inspire de BOTL (Rensink et al., 2002), une logique générique orientée-objet (voir aussi chapitre 2, section 2.4). Vérifier des contraintes OCL avec BOTL exige la traduction d'OCL vers cette logique. La logique que nous proposons est plutôt une extension d'OCL, exprimée en deux niveaux. Dans le premier niveau, on retrouve les *propriétés d'états* représentées par des expressions OCL auxquelles on ajoute des prédicats spécifiquement orientés-objet. Dans le deuxième niveau, nous avons les *formules temporelles* ayant comme base la logique CTL (Clarke et al., 1984), enrichie par les quantificateurs existentiels et universels bornés.

4.4.1 Syntaxe d'OCL^{EXT}

Soit P_{exp} l'ensemble de propriétés d'états et F_{exp} l'ensemble de formules temporelles de la logique. La syntaxe de cette logique est donnée récursivement comme suit:

$$e(\in P_{exp}) ::= x \mid v \mid e.\mathtt{a} \mid \omega(e, \ldots e) \mid e\,.\,\textbf{iterate}\ (x_1\ ;\ x_2 = e \mid e\ ) \mid e\ @\textbf{pre}$$
$$\mid e.\textbf{owner} \mid \textbf{act}(e)$$
$$\varphi(\in F_{exp}) ::= e \mid \neg\phi \mid \phi \wedge \psi \mid \forall^{<l} z \vdash \tau : \phi \mid \textbf{EX}\phi \mid \textbf{E}[\phi \textbf{U} \psi] \mid \textbf{A}[\phi \textbf{U} \psi]$$

Propriétés d'états

- x, v, $e.\mathtt{a}$, $\omega(e, \ldots e)$, e . **iterate** et e@**pre** ont les mêmes significations que dans les expressions OCL
- **act**(e) indique si l'objet e est actif ou si la méthode e est active. Un objet est actif de sa création à sa destruction. Une méthode est active de son appel (elle se retrouve dans la pile d'appels) à son retour (la valeur de retour est $\neq \perp$, et elle disparaît de la pile d'appels dans l'état suivant).

- e.**owner** dénote l'objet qui exécute la méthode e.

On dénote par B_{exp} l'ensemble des propriétés d'état booléennes:

$$B_{exp} = \{e \in P_{exp} \mid \vdash e : \mathsf{Bool}\}$$

Formules temporelles Une formule se construit sur la base des propriétés d'états booléennes, des opérateurs logiques de premier ordre ($\neg, \wedge$, etc.), des opérateurs temporaux CTL et des quantificateurs. Deux types de connecteurs temporaux sont utilisés en combinaison:

1. **A** et **E** qui signifient respectivement *pour tout chemin* et il *existe un chemin*
2. **X**, **F**, **G** et **U** qui signifient respectivement *dans le prochain état, dans un futur état, dans tout futur état*, et *jusqu'à (until)*

Les prédicats de bases sont donnés par les expressions `OCL` booléennes. Plus précisément on a:

- e est une propriété d'état booléenne ($e \in B_{exp}$).
- $\mathbf{EX}\phi$ exprime qu'il existe un état suivant dans lequel la formule ϕ est vraie.
- $\mathbf{E}[\phi \mathbf{U} \psi]$ exprime qu'il existe un chemin à partir de l'état courant sur lequel on arrive à un état où ψ est vraie, et dans chaque état d'avant ϕ est vraie.
- $\mathbf{A}[\phi \mathbf{U} \psi]$ exprime que pour tout chemin qui part de l'état courant, on arrive dans un état où ψ est vraie, et dans chaque état précédent ϕ est vraie.
- On dénote: $z \vdash \tau$ pour $\vdash z : \tau$. Alors,
  $\forall^{<l} z \vdash \tau : \phi$ représente le quantificateur universel borné par l, un entier, et il

exprime que la formule ϕ doit être vraie pour chaque instance du type τ, où le nombre d'instances de chaque type est limité à une constante $l \in \mathbb{N}$, de la façon suivante:

1. si $z \vdash \mathsf{Int}$ alors $z < l$
2. si $z = (C, i) \vdash \mathtt{Obj}^C$ alors $i < l$
3. si $z = (o, M, j) \vdash \mathtt{Met}^{C,M}$ alors $j < l$

Cette restriction est exigée pour éviter la création d'espaces d'état infinis dû au fait que le domaine des valeurs des types est généralement infini.

4.4.2 Sémantique d'$OCL^{\mathtt{EXT}}$

4.4.2.1 Chemin

Pour exprimer la sémantique de $\mathrm{OCL}^{\mathtt{EXT}}$ on a besoin de la notion de chemin. Les modèles UML et leur graphe d'exécution possèdent des états finaux. Ceci implique en particulier que la relation de transitions n'est pas forcément totale, et par conséquent, on peut rencontrer des problèmes pour exprimer certaines propriétés (qui contiennent des opérateurs temporaux tels que **EX**). Pour éliminer cet inconvénient à chaque état final on ajoute un boucle.

Soit $\mathcal{S}_F \subset \mathcal{S}$, l'ensemble des états finaux, alors:
$\forall s_f \in \mathcal{S}_F$ on a $(s_f, s_f) \in \mathcal{R}$

Nous obtenons ainsi des chemins infinis. Un chemin infini $c = s_0 s_1 s_2 \cdots$ est une séquence infinie d'états $(s_0, s_1, s_2, \cdots)$, tel que $\forall i (s_i, s_{i+1}) \in \mathcal{R}$.

$c[i] = s_i$ représente le $i + 1$ élément du chemin. L'ensemble des tous les chemins

partant de l'état initial $s_0 = (\sigma_0, \gamma_0)$ est défini:

$$\mathcal{P}_{\mathcal{K}}(\sigma_0, \gamma_0) = \{c \in \mathcal{S}^{\omega} \mid c(0) = (\sigma_0, \gamma_0), \forall i \geqslant 0 : (c[i], c[i+1]) \in \mathcal{R}\}$$

où $\mathcal{S}^{\omega}$ dénote l'ensemble de séquences d'états finies ou infinies.

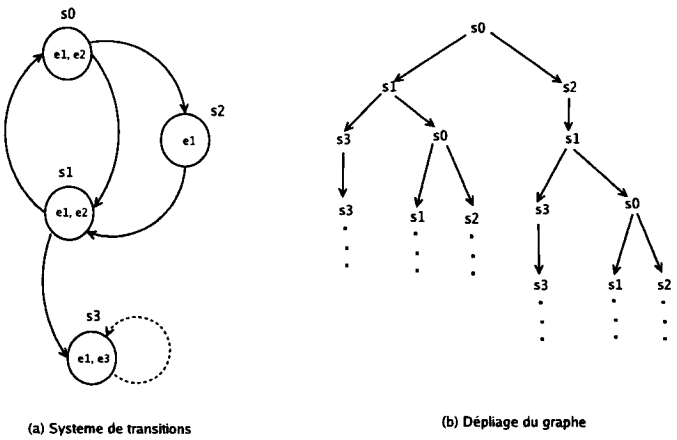

Figure 4.8 Exemple: graphe et dépliage

Exemple 4.11 La figure 4.8 (a) illustre l'exemple d'un graphe d'exécution et dans (b) on retrouve le dépliage correspondant à ce graphe. La relation de transitions pour ce graphe est: $\mathcal{R} = \{(s_0, s_1), (s_1, s_0), (s_0, s_2), (s_2, s_1), (s_1, s_3), \}$. L'état s_3 est un état final. Pour rendre la relation de transitions totale on ajout l'élément (s_3, s_3). Par conséquent tout chemin partant de l'état initial est infini: $c_1 = s_0 s_1 s_3 s_3 \ldots$, $c_2 = s_0 s_1 s_0 s_1 \ldots$, etc.

4.4.2.2 Sémantique de P_{exp}

La sémantique de cette logique est exprimée sur la structure étendue $\mathcal{K}_{\Upsilon} =< \mathcal{S} \times \Upsilon, \mathcal{R}_{\Upsilon} >$ (voir chapitre 3, section 5.2).

Les propriétés d'états sont exprimées à partir des expressions OCL dont la sémantique est donnée dans la section 4.3. La sémantique des éléments orientés-objet est donnée par la fonction:

$$[\![\_]\!] : P_{exp} \rightarrow ((\Sigma \times \Gamma) \times \Upsilon \times \Theta \times MId) \rightarrow Val_{\mathtt{OCL}}^{\perp}$$

où $\Sigma, \Gamma, \Upsilon, \Theta$ et MId ont les mêmes significations que dans la section 4.3.

- $[\![e.\mathbf{owner}]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = o$ où $[\![e]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = (o, M, j)$
- $[\![\mathbf{act}(e)]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} = [\![e]\!]^{\theta,m}_{\sigma,\gamma,\varepsilon} \in (dom(\sigma) \cup dom(\gamma))$

4.4.2.3 Sémantique de F_{exp}

La sémantique d'une formule logique temporelle est donnée par la relation de satisfaction:

$$\models \subseteq ((\Sigma \times \Gamma) \times \Upsilon \times \Theta \times MId) \times F_{exp}$$

Soit $(\sigma_0, \gamma_0) \in \mathcal{S}$ l'état initial du système de transitions $\mathcal{K}$. $\mathcal{P}_{\mathcal{K}}(\sigma_0, \gamma_0)$ l'ensemble des chemins partant de (σ_0, γ_0) et $\theta \in \Theta$ un environnement de variables. La relation $\models$ est définie inductivement comme suit:

$$
\begin{array}{lcl}
\sigma_0, \gamma_0, \theta \vDash e & \Longleftrightarrow & [\![e]\!]^{\theta,m}_{\sigma_0,\gamma_0,\varepsilon_0} = \mathsf{True} \\
\sigma_0, \gamma_0, \theta \vDash \neg\phi_1 & \Longleftrightarrow & \sigma_0, \gamma_0, \theta \nvDash \phi_1 \\
\sigma_0, \gamma_0, \theta \vDash \phi_1 \wedge \phi_2 & \Longleftrightarrow & (\sigma_0, \gamma_0, \theta \vDash \phi_1) \text{ et } (\sigma_0, \gamma_0, \theta \vDash \phi_2) \\
\sigma_0, \gamma_0, \theta \vDash \forall^{<l} z \vdash \tau : \phi & \Longleftrightarrow & \sigma_0, \gamma_0, \theta[z \mapsto v] \vDash \phi \text{ pour tout } v \in Val^{\tau} \text{ t.q.} \\
 & & \quad 1.\ \text{si } z \vdash \mathsf{Int} \text{ alors } z < l \\
 & & \quad 2.\ \text{si } z \vdash \mathsf{Obj}^{C} \text{ et } z = (C, i) \text{ alors } i < l \text{ et} \\
 & & \quad v \in dom(\sigma_0) \\
 & & \quad 3.\ \text{si } z \vdash \mathsf{Met}^{C,M} \text{ et } z = (o, M, j) \text{ alors} \\
 & & \quad i < l \text{ et } v \in dom(\gamma_0) \\
\sigma_0, \gamma_0, \theta \vDash EX\phi_1 & \Longleftrightarrow & \exists c \in \mathcal{P}_{\mathcal{K}}(\sigma_0, \gamma_0) : c[1], \theta \vDash \phi_1 \\
\sigma_0, \gamma_0, \theta \vDash E[\phi_1 U \phi_2] & \Longleftrightarrow & \exists c \in \mathcal{P}_{\mathcal{K}}(\sigma_0, \gamma_0) : \\
 & & \exists j \geq 0 : c[j], \theta \vDash \phi_2 \wedge \forall\ 0 \leq k < j : c[k], \theta \vDash \phi_1 \\
\sigma_0, \gamma_0, \theta \vDash A[\phi_1 U \phi_2] & \Longleftrightarrow & \forall c \in \mathcal{P}_{\mathcal{K}}(\sigma_0, \gamma_0) : \\
 & & \exists j \geq 0 : c[j], \theta \vDash \phi_2 \wedge \forall\ 0 \leq k < j : c[k], \theta \vDash \phi_1
\end{array}
$$

4.4.2.4 Abréviations et équivalences

On utilise les abréviations suivantes:

- $\forall^{<l} z \in \mathbf{act}(\tau) : \phi$ à la place de $\forall^{<l} z \vdash \tau : \mathbf{act}(z) \Rightarrow \phi$. Elle exprime que pour tout objet (méthode) actif de type τ, ϕ est vrai.
- $\forall^{<l} z \in e.\mathsf{Met} : \phi$ (où $e \vdash \mathsf{Obj}^{C}$) à la place de $\forall^{<l} z \vdash \mathsf{Met}^{C} : (z.\mathbf{owner} = e) \Rightarrow \phi$, ce qui signifie que pour chaque instance de méthode qui est executée par un objet donné, ϕ est valide.

Voici les équivalences les plus importantes des connecteurs CTL, quantificateurs et implication qu'on peut employer dans la syntaxe de formules:

1. $\mathbf{AX}(\phi) \equiv \neg\mathbf{EX}(\neg\phi)$ signifie que pour tous les prochains états ϕ est vérifiée.

2. $\mathbf{AF}(\phi) = \mathbf{A}[\mathsf{True}\mathbf{U}\phi]$ signifie que pour tous les chemins, dans le futur ϕ est vérifiée.

3. $\mathbf{EF}(\phi) = \mathbf{E}[\mathsf{True}\mathbf{U}\phi]$ signifie qu'il existe au moins un chemin pour lequel, dans futur état, ϕ est vérifiée.

4. $\mathbf{AG}(\phi) \equiv \neg\mathbf{EF}(\neg\phi)$, signifie que ϕ est vérifiée dans tous les états du système (ϕ est un invariant).

5. $\mathbf{EG}(\phi) \equiv \neg\mathbf{AF}(\neg\phi)$ signifie qu'il existe un chemin pour lequel ϕ est vérifiée pour chaque état.

6. $\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$

7. $\forall^{<l} z \vdash \tau : \phi \equiv \neg\exists^{<l} z \vdash \tau : \neg\phi$

La figure 4.9 illustre graphiquement la signification de certains connecteurs CTL.

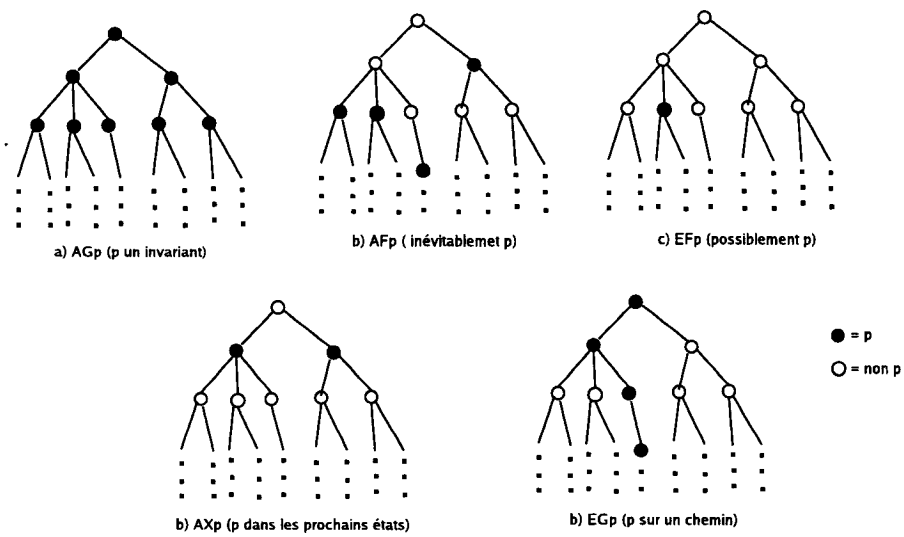

Figure 4.9 Sémantique des opérateurs: *AG*, *AF*, *EF*, *AX* et *EG*

4.5 Contraintes

Les contraintes sont des conditions que le système doit respecter. La syntaxe d'une contrainte est exprimée sous la forme d'un patron qui contient le contexte et une expression OCL booléenne. OCL standard définit trois types de contraintes: l'invariant et les pré/post-conditions. Chaque contrainte est exprimée sur un contexte. Pour l'invariant le contexte est une classe et pour les pré/post-conditions le contexte est une méthode. On dénote par C_{exp} l'ensemble de ces contraintes. La syntaxe de contraintes est donnée comme suit:

$$c(\in C_{exp}) \quad ::= \quad \textbf{context } C \textbf{ inv } e \quad | \quad \textbf{context } C :: M(\vec{p}) \textbf{ pre } e_1 \textbf{ post } e_2$$

où

- C est la classe sur laquelle l'invariant est exprimé et e est l'expression OCL de l'invariant
- $M(\vec{p})$ est la méthode de la classe C sur laquelle la paire pré/post-condition est exprimée. $\vec{p}$ représente l'ensemble de paramètres formels de la méthode. e_1, e_2 sont les expressions OCL qui exprime respectivement la pré-condition et la post-condition.

La sémantique des contraintes OCL est exprimée à l'aide de OCL^{EXT}.

4.5.1 Invariant

Un invariant est une propriété qui doit être vérifiée dans chaque état, contenant des objets de la classe du contexte et aucun d'eux n'exécute pas une méthode.

$$\textbf{context } C \textbf{ inv } e \equiv$$
$$\mathbf{AG}[\forall^{<l} z \in \mathbf{act}(\mathsf{Obj}^C) : ((\forall^{<l} m_1 \in z.\mathsf{M}_1 : \ldots \forall^{<l} m_k \in z.\mathsf{M}_k) :$$
$$(\neg\mathbf{act}(m_1) \wedge \ldots \neg\mathbf{act}(m_k)) \Rightarrow e]$$

où

- z est un objet actif de la classe C.
- M_i, $(1 \leqslant i \leqslant k)$ sont les signature de méthodes de la classes C; $dom(C.meths)$ $= \{\mathsf{M}_1, \ldots \mathsf{M}_k\}$.
- m_i, $(1 \leqslant i \leqslant k)$ sont les instances de la méthode M_i, exécutées par l'objet z.

4.5.2 Pré/post-condition

Une pré/post-condition est vérifiée si et seulement si pour toute instance de la méthode, exécutée par un objet de sa classe, nous avons: si la pré-condition est vraie au moment où la méthode est appelée, alors la post-condition doit être vraie dans chaque état de retour.

$$\textbf{context } C :: M(\vec{p}) \textbf{ pre } e_1 \textbf{ post } e_2 \equiv$$
$$\forall^{<l} z \in \mathbf{act}(\mathsf{Obj}^C) : \forall^{<l} m \in z.\mathsf{M} :$$
$$\mathbf{AG}[call(m) \Rightarrow \mathbf{AX}[\mathbf{AG}[return(m)] \Rightarrow e_2]]]$$

où

- $call(m) = \neg act(m) \wedge \mathbf{EX}[act(m) \wedge e_1]$
- $return(m) = act(m) \wedge \mathbf{AX}[\neg act(m)]$
- z est un objet actif de la classe C.
- m est une instance de la méthode M exécutée par l'objet z ($M \in dom(C.meths)$).

4.5.3 Autres contraintes

Les contraintes OCL ne sont pas limitées aux invariants et pré/post-conditions. Cette logique permet d'exprimer n'importe quelle propriété de vivacité. Par exemple le patron **after/eventually**.

Informellement, il exprime que si e_1 est vraie dans un état, alors e_2 sera vraie dans un état futur, pour tous les chemins.

$$\textbf{context } C \textbf{ after } e_1 \textbf{ eventually } e_2 \equiv$$
$$\mathbf{AG}[e_1 \Rightarrow \mathbf{A}[\mathsf{True}\mathbf{U}e_2]]$$

4.6 Vérification à la volée

4.6.1 Algorithme LMCO

LMCO (Local Model Checking for OCL^{EXT}) est une adaptation d'**ALMC** (Local Model Checking for CTL), de B. Vergauwen et J. Lewi (B. Vergauwen and J. Lewi, 1993) à OCL^{EXT}.

L'algorithme LMCO donné à la figure 4.10 prend en entrée le système de transitions

$\mathcal{K}_\Upsilon$, la formule OCL^{EXT}, ϕ et l'état initial s_0. Il doit satisfaire la spécification suivante:

$$func\ \texttt{LMCO}(\mathcal{K}_\Upsilon, s_0, \phi)\ \text{return}\ \texttt{True}\ \text{ssi}\ \mathcal{K}_\Upsilon, s_0 \models \phi.$$

On dit alors que le système est un modèle de la formule.

Une formule est composée par un ensemble de sous-formules créées récursivement sur la syntaxe de la logique. Elle est évaluée dans les états du système de transitions par induction des ses sous-formules. L'information de chaque sous-formule sur le parcours du système de transitions est conservée dans la variable globale *info*:

$$info : \mathcal{S} \times sub(\phi) \rightarrow \{0, 1, \perp\}$$

où $sub(\phi)$ représente l'ensemble des sous-formules de ϕ.
Soit $\varphi \in sub(\phi)$, une sous-formule de ϕ. Informellement, on a: $info(s, \varphi) = 0$ si l'état s ne satisfait pas la sous-formule φ, $info(s, \varphi) = 1$ si l'état s satisfait la sous-formule φ et $info(s, \varphi) = \perp$ si on ne peut pas se prononcer à ce stade.

L'invariant de l'algorithme $\texttt{LMCO}(\mathcal{K}, s_0, \phi)$ est:

$$[I = \forall s, \varphi :\ (info(s, \varphi) = 0) \Rightarrow \mathcal{K}_\Upsilon, s \not\models \varphi) \wedge (info(s, \varphi) = 1 \Rightarrow \mathcal{K}_\Upsilon, s \models \varphi)]$$

Le temps de calcul de l'algorithme dépend de la longueur de la formule $|\phi|$ et de la taille du système de transition $|\mathcal{K}_\Upsilon| = |\mathcal{S}| + |\mathcal{R}_\Upsilon|$. Si on fait abstraction de la taille de la relation de transitions, on a que le temps de calcul de l'algorithme LMCO est $|\phi|.|\mathcal{S}|$. La taille d'une formule est calculée par induction sur la structure syntaxique de la logique OCL^{EXT}: $|e| = 1$, $|\neg\phi_1| = |EX\phi_1| = |AX\phi_1| = 1 + |\phi_1|$, $|\phi_1 \wedge \phi_2| = |E[\phi_1 \mathrm{U} \phi_2]| = |A[\phi_1 U \phi_2]| = 1 + |\phi_1| + |\phi_2|$. Il est à noter qu'on fait

```
func LMCO(K, s_0, φ) return Bool is
   var info : S × sub(φ) → {0, 1, ⊥};
   var env : Var_OCL ⇀ Val_OCL;

   proc Check(s, φ) is
   /* pre: I */
   /* post: I ∧ info(s, φ) ≠⊥ */
            proc CheckEU(s_r, φ_1, φ_2)
            /*  pre: I ∧ info(s_r, E[φ_1 U φ_2]) =⊥ */
            /*  post: I ∧ info(s_r, E[φ_1 U φ_2]) ≠⊥ */
            proc CheckAU(s_r, φ_1, φ_2)
            /*  pre: I ∧ info(s_r, A[φ_1 U φ_2]) =⊥ */
            /*  post: I ∧ info(s_r, A[φ_1 U φ_2]) ≠⊥ */

   begin / * Check * /
            if info(s, φ) ≠⊥ then return fi;
            case φ is
            e ∈ B_exp :      if s = (σ, γ), où [[e]]^{θ,m}_{σ,γ,ε} then info(s, φ) := 1 else
                             info(s, φ) := 0 fi;
            ¬φ_1 :           Check(s, φ_1);
                             if info(s, φ_1) = 1 then info(s, φ) := 0 else
                             info(s, φ) := 1 fi;
            φ_1 ∧ φ_2 :      Check(s, φ_1);
                             if info(s, φ_1) = 0 then info(s, φ) := 0 return fi;
                             Check(s, φ_2); info(s, φ) := info(s, φ_2)
            EX φ_1 :         foreach s' such that sRs' do
                                 Check(s', φ_1);
                                 if  info(s', φ_1) = 1 then info(s, φ) := 1; return fi
                             od;
                             info(s, φ) := 0
            AX φ_1 :         foreach s' such that sRs' do
                                 Check(s', φ_1);
                                 if  info(s', φ_1) = 0 then info(s, φ) := 0; return fi
                             od;
                             info(s, φ) := 1
            ∀^{<l} z ⊢ τ : φ :  foreach o ∈ dom_σ(τ) ∧ i < l do
            / * dom_σ(τ) est le domaine d'objets actifs de type τ dans l'état courant
             et o = (C, i) * /
                             env[z ↦ o];
                             Check(s, φ);
                                 if  info(s, φ) = 0 then info(s, φ) := 0; return fi
                             od;
                             info(s, φ) := 1
            E[φ_1 U φ_2] :   CheckEU(s, φ_1, φ_2);
            A[φ_1 U φ_2] :   CheckAU(s, φ_1, φ_2);
            esac
   end; / * Check * /
begin / * LMCO * /
   Check(s_0, φ);
end
```

Figure 4.10 Algorithme à la volée

```
(1)  proc CheckEU(s_r, φ1, φ2) is
(2)     var goal : S ∪ {⊥}
(3)     proc Visit(s) is
(4)     begin
(5)          marquer l'état s
(6)          case info(s, E[φ1 U φ2]) is
(7)          0 : return;
(8)          1 : goal := s; goto π
(9)          ⊥: Check(s, φ2); if  info(s, φ2) = 1 then goal := s; goto π fi;
(10)            Check(s, φ1); if info(s, φ1 = 0) then return fi;
(11)            /* info(s, E[φ1 U φ2] =⊥) ∧ (K, s ⊨ φ1) ∧ (K, s ⊭ φ2) */
(12)            foreach s' such that sRs' do
(13)               marquer la transition (s, s');
(14)               if pas marquée then
(15)                  Visit(s');
(16)               fi;
(17)            od;
(18)         esac
(19)    end;  /* Visit */
(20) begin /* CheckEU */
(21) goal :=⊥;  Visit(s_r);
(22) π:      if goal =⊥ then info(s_r, E[φ1 U φ2]) := 0 else info(s_r, E[φ1 U φ2]) := 1 fi;
(23) end
```

Figure 4.11 Procédure CheckEU naïve

abstraction de la complexité de l'évaluation des propriétés d'état auxquelles on attribue un temps unitaire.

4.6.2 Procédure CheckEU naive

La procédure *CheckEU naive* (figure 4.11) est simple, elle suit la sémantique de l'opérateur $E[\_U\_]$. Sa complexité est toutefois quadratique.

Selon cet algorithme, la vérification d'une telle formule débute par la deuxième partie d'*until* (ligne (9), figure 4.11). Si φ_2 est vrai dans l'état courant, la formule est vraie; on arrête alors la construction du graphe et on retourne la réponse (*goto* π). Si φ_2 est fausse, on vérifie la première partie (φ_1). Si elle est aussi fausse (ligne

(10)), la formule est fausse, on arrête alors la construction du graphe et on retourne la réponse.

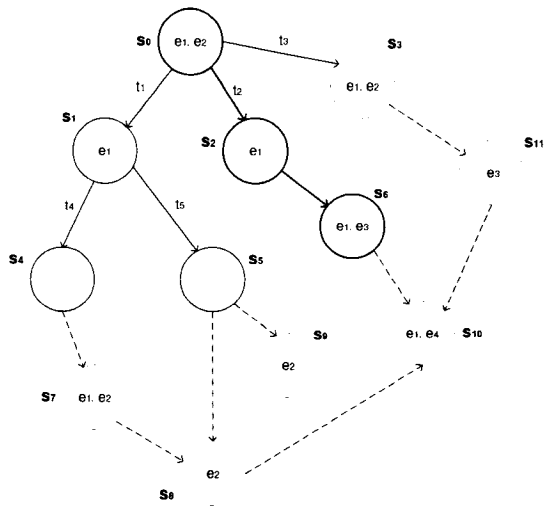

Figure 4.12 Vérification de $E[\varphi_1 U \varphi_2]$

Exemple 4.12 Pour illustrer ce propos, on se réfère au graphe illustré à la figure 4.12. Chaque état est décoré par l'ensemble des expressions OCL booléennes vraies dans l'état. On se propose de vérifier la formule $\phi = E[e_1 U e_3]$. On commence la vérification dans l'état initial, s_0, où, on ne peut pas se prononcer pour ϕ, car e_3 =False mais, e_1 =True. On continue le calcul du graphe. Les transitions sortantes t_1, t_2, t_3 sont marquées et le prochain état visité est s_1. On se retrouve dans la même situation qu'à l'état initial (e_3 = False, e_1 = True), donc pas de réponse pour ϕ. Les transitions t_4, t_5 sont marquées et le prochain état visité est s_4, où ni e_3 ni e_1 ne sont présentes, et donc ϕ = False. On arrête la construction pour tout chemin qui sort de s_4. On retourne à l'état s_1 et comme il y a une autre transition marquée, on effectue la vérification dans son état cible. Pour cet exemple le prochain état visité est s_5. Comme e_3 = False et aussi e_1 = False, alors ϕ = False, on arrête pour cette branche, car il n'y a pas une autre transition sortante de s_1. De la même manière, on continue la recherche en profondeur. Selon l'exemple, la vérification du graphe, (et implicitement la construction) s'arrête

définitivement dans l'état s_6, car on est en mesure de donner une réponse pour ϕ. Cette formule est vérifiée sur le chemin $c = s_0 s_2 s_6$.

Il faut noter la caractéristique d'algorithme à la volée que la partie en pointillé n'a pas été construite, en utilisant cette méthode alternative, contrairement à l'algorithme basé sur le μ-calcul de la première version de SOCLe. Dans cet exemple un graphe de petite taille a été présenté. Imaginons qu'on doive calculer un graphe plus grand que la capacité de la mémoire, et alors vérification ne sera jamais effectuée. Par contre, en adoptant la vérification à la volée, pour un nombre important de cas, on est capable de fournir la réponse assez rapidement (après 6 états, dans l'exemple). Cet algorithme est efficace surtout quand la formule est falsifiée ou vérifiée sur un préfixe du graphe. Dans le pire des cas le graphe est construit au complet. Par exemple, pour la formule $E[\mathsf{True}\ U\ \mathsf{False}]$. Voici toutefois une exemple où le temps de calcul est quadratique:

Exemple 4.13 Soit, toujours sur l'exemple de la figure 4.12 :
$\phi = E[\mathsf{True}\ U\ E[\mathsf{True}\ U\ \mathsf{False}]]$, et on dénote $\varphi_1 = \mathsf{True}$ et $\varphi_2 = E[\mathsf{True}\ U\ \mathsf{False}]$.

- état s_0 : $\varphi_2 = \mathsf{False}$, $\varphi_1 = \mathsf{True}$, $\phi = \perp$, temps de calcul: $|\mathcal{S}|$.
- état s_1 : $\varphi_2 = \mathsf{False}$, $\varphi_1 = \mathsf{True}$, $\phi = \perp$, temps de calcul: $|\mathcal{S}| - 1$.
- ...
- état s_i : $\varphi_2 = \mathsf{False}$, $\varphi_1 = \mathsf{True}$, $\phi = \perp$, temps de calcul: $|\mathcal{S}| - i$.
- ...
- état $s_1 1$: $\varphi_2 = \mathsf{False}$, $\varphi_1 = \mathsf{True}$, $\phi = \perp$, temps de calcul: 0.

Donc, pour les formules *until* imbriquées le temps de calcule peut être quadratique: $\mathcal{O} = \sum_{i=0}^{n} |\mathcal{S}| - i = \sum_{i=0}^{n} = n(n-1)/2 = \mathcal{O}(n^2)$ où $|\mathcal{S}| = n$.

4.6.3 Procédure CheckAU

Informellement la procédure $CheckAU[\varphi_1 U \varphi_2]$ exprime que pour tout chemin partant de l'état courant on doit arriver dans un état où φ_2 est vraie, et dans tous les états précédents, φ_1 est vraie. Cet algorithme, présenté à la figure 4.13, est beaucoup plus simple que le $CheckEU$. Il suffit de détecter un chemin qui ne vérifie pas la formule (ligne 6), ou un cycle (π_{cycle}, ligne 13) tel que dans chaque état φ_1 est vraie mais jamais φ_2. On conclut que le chemin ne vérifie pas $A[\varphi_1 U \varphi_2]$ et donc $CheckAU[\varphi_1 U \varphi_2]$ est aussi fausse.

```
(1)  proc CheckAU(s_r, φ1, φ2) is
(2)     var cp : S*; /* le chemin courant de recherche (une pile)*/
(3)     proc Visit(s) is
(4)     begin
(5)        case info(s, A[φ1 U φ2]) is
(6)        0 : goto π;
(7)        1 : return;
(8)        ⊥: Check(s, φ2); if  info(s, φ2) = 1 then  info(s, A[φ1 U φ2]) := 1; return fi;
(9)           Check(s, φ1); if  info(s, φ1) = 0 then  info(s, A[φ1 U φ2]) := 0;  goto π fi;
(10)          /* (info(s, A[φ1 U φ2] =⊥) ∧ (M, s ⊨ φ1) ∧ (M, s ⊭ φ2) */
(11)          empiler l'état s dans cp
(12)          foreach  s' such that sRs' do
(13)             if s' ∉ cp  then Visit(s') else {π_cycle} goto π fi
(14)          od
(15)          info(s, A[φ1 U φ2]) := 1;
(16)          depiler s de cp;
(17)       esac
(18)    end;  /* Visit */
(19) begin /* CheckAU */
(20)    cp := null;  Visit(s_r);
(21) π:      foreach s ∈ cp do info(s, A[φ1 U φ2]) := 0 od
(22) end
```

Figure 4.13 Procédure CheckAU

Exemple 4.14 Pour le graphe de la figure 4.12 la formule $\phi = A[e_1 U e_3]$ est fausse pour le chemin $c = s_0 s_1 s_2$. Donc, après trois états on est en mesure de répondre.

4.6.4 Linéarisation de l'algorithme naïf

La source de la non linéarité est donc la présence des formules $E[\_U\_]$ imbriquées. Reprenons l'exemple de deux formules *until* imbriquées: $\phi = E[\varphi_1 U \varphi_2]$ où $\varphi_2 = E[\varphi_3 U \varphi_4]$. La fonction $CheckEU(s_r, \varphi_1, \varphi_2)$ nous retourne $info(s_r, E[\varphi_1 U \varphi_2]) = 1$ ssi $\mathcal{K}_\Upsilon, s_r \vDash E[\varphi_3 U \varphi_4]$. Donc, on a l'information complète de la formule ϕ dans l'état s_r. Cette information, est obtenu par $CheckEU(s_r, \varphi_3, \varphi_4)$ qui peut devoir parcourir le système de transitions entièrement. Alors, l'idée est de conserver le résultat de $CheckEU(s_r, \varphi_3, \varphi_4)$. On a donc besoin d'un critère de décidabilité de la formule $E[\varphi_1 U \varphi_2]$ dans chaque état π visité.

Le lemme 4.1 assure la correction d'une version récursive de $CheckEU()$, qui accumule de l'information sur la formule, sans un retour en arrière.

Lemme 4.1 Pour chaque état $s \in \mathcal{S}_{eu}$ on a:

$$[1]\ \ (\pi.goal = \bot) \Rightarrow (\mathcal{K}_\Upsilon, s \nvDash E[\varphi_1 U \varphi_2])$$

$$[2]\ \ (\pi.goal \neq \bot) \Rightarrow (\mathcal{K}_\Upsilon, s \vDash E[\varphi_1 U \varphi_2]\ ssi\ \ s \tilde{\rightsquigarrow} \pi.goal)$$

où $\rightsquigarrow$ est l'ensemble de transitions marquées par $CheckEU(s_r, \varphi_1, \varphi_2)$ et $\tilde{\rightsquigarrow}$ est la fermeture réflexive, transitive de $\rightsquigarrow$.

Conformément à ce lemme la procédure $CheckEU()$ est modifiée en conséquence (voir figure 4.14). La linéarisation de l'algorithme au complet est détaillée à l'annexe I.1.

Exemple 4.15 En reprenant l'exemple 4.13 avec l'algorithme linéaire on a que la complexité de calcul est $|\mathcal{S}|$. La raison est qu'on a l'information nécessaire pour conclure dans l'état s_0. Aucun état du système n'est vérifie pas $\varphi_2 = E[\text{True } U \text{ False}]$.

```
proc CheckEU(s_r, φ1, φ2) is
    var goal : S ∪ {⊥}
    proc Visit(s) is begin /* pareil qu'avant */ end;

begin /* CheckEU */
    goal :=⊥; Visit(s_r);
π:  if goal :=⊥ then /* on utilise lemme [1] */
        foreach état s marqué do info(E[φ1 U φ2]) := 0 od
    else /* goal ∈ S, on utilise lemme [2] */
        foreach état s marqué do
            if s⇝ goal then info(E[φ1 U φ2]) := 1 else info(E[φ1 U φ2]) := 0 fi
    fi
end
```

Figure 4.14 Procédure CheckEU linéaire

Les deux algorithmes: naïf et linéaire, sont fonctionnels dans l'outil SOCLe et ont été implantés en `OCAML`.

CHAPITRE 5

ÉTUDE DE CAS

Ce chapitre présente une étude de cas qui permet premièrement de montrer comment modéliser et vérifier un système à l'aide de l'outil SOCLe et deuxièmement de valider la technique de vérification à la volée. Cette étude de cas n'est pas simplement un exemple académique. Elle est l'objet d'un papier (Painchaud et al., 2005) écrit en collaboration avec RDDC.

5.1 Introduction

L'objectif principal de cette étude de cas est de modéliser et vérifier les *Cavéats*, aussi connus sous le nom de *Multi-Level Security.*

Les *Cavéats* représentent des *catégories de sécurité*, qui s'additionnent aux restrictions de classifications sécurisées existantes des documents. Elles sont importantes pour le Ministère National de la Défense (Departement of National Defense - DND) et leur Système de Contrôle (Command and Control Information Systems - C2IS). Certains C2IS travaillent dans le mode *System High Mode*, c'est-à-dire que chaque personne qui a accès au système a la possibilité de voir l'information du système. Or, tout le monde n'est pas obligé de tout savoir. Pour limiter les droits d'accès, le *Multi-Level Security* est proposé. Les types de cavéats les plus communs consistent en une restriction par nation (i.e. Canadien Eyes Only (CEO), Canadian or US Eyes Only (CANUS)). Un cavéat tout seul ne signifie rien. Il doit être attaché à un niveau de sécurité. Par exemple, un document peut être étiqueté *Secret CEO*

or *Top Secret CEO* ou quelque chose de similaire.

5.2 Survol sur la modélisation

On modélise des usagers qui essaient d'accéder aux documents sécurisés. Un contrôleur surveille l'accès. Les données d'entrées sont stockées en trois tableaux (5.1, 5.5, 5.6 - section 5.3). Ils représentent les contraintes imposées au système. La spécification informelle au complet se retrouve dans (Painchaud et al., 2004). Elle sera présentée au fur et à mesure de la modélisation.

L'étude de cas des Cavéats comporte trois parties. Dans la première partie, on expose une façon de représenter les tableaux de données. Pour cette partie, il faut spécifier surtout la structure statique du système. On utilise l'héritage, supporté par notre outil, pour mieux structurer les données. Le comportement du système est simple (figure 5.1): un usager demande l'accès à un document. Le document est confondu avec le contrôleur.

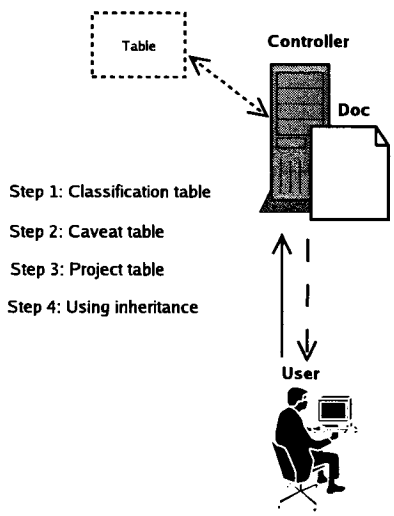

Figure 5.1 Modélisation des tables

Dans la deuxième partie, schématisée à la figure 5.2, on présente une forme plus complexe du système de Cavéats en introduisant le contrôleur. Celui-ci joue deux rôles: il contrôle le protocole d'accès aux documents et il administre les données d'entrées (fait des changements dans les tableaux). Un comportement dynamique plus important sera spécifié.

La dernière partie est destinée à la vérification des contraintes et à l'analyse de l'étude de cas.

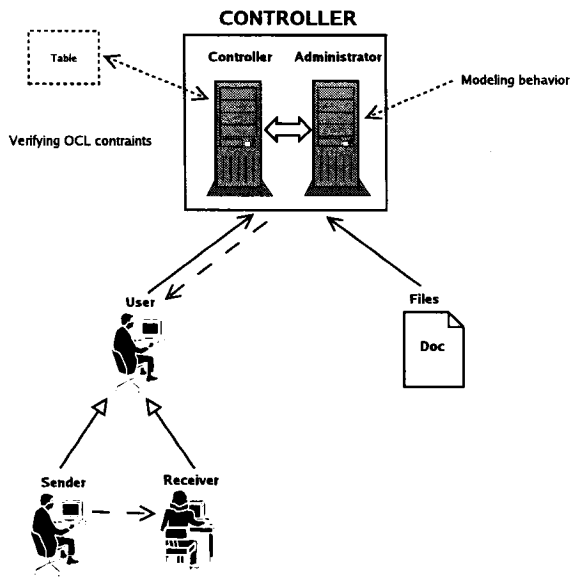

Figure 5.2 Modélisation du comportement

5.3 Modélisation des tables

Cette section explique comment encoder un tableau de valeurs à l'aide des objets UML et des expressions OCL. Cela représente un choix de modélisation. Pour cette raison, dans cette partie, il faudra insister plutôt sur la structure statique du système: le diagramme de classes et le diagramme d'objets.

5.3.1 Étape 1 - Tableau Classification

On prend comme point de départ l'étude de cas sous la forme réduite (figure 5.1). Caractérisé par un niveau de visibilité, l'usager essaie d'accéder à un document protégé ou non. S'il est protégé, un niveau de sécurité le caractérise. La politique d'accès est représentée dans le tableau 5.1 et doit être vue comme une contrainte stricte imposée au système. Comment lire une tel tableau? Par exemple, un document classé *Secret* ne peut être consulté que par un usager qui a une visibilité *Secret* ou *TopSecret*.

Tableau 5.1 Tableau Classification

Niv. classif./Sec. du doc.	**Enhanced**	**Confidential**	**Secret**	**TopSecret**
Unclassified	*Granted*	*Granted*	*Granted*	*Granted*
Confidential	*Denied*	*Granted*	*Granted*	*Granted*
Secret	*Denied*	*Denied*	*Granted*	*Granted*
TopSecret	*Denied*	*Denied*	*Denied*	*Granted*

La configuration statique du système est représentée par le diagramme de classes de la figure 5.3. Il comporte quatre classes:

- La classe `User` présente deux champs: *clearance*, un entier qui représente le niveau de visibilité d'un usager, et *accessRead*, un booléen qui indique si l'usager a obtenu l'accès au document. Cette classe permet la création des usagers.

- La classe `Document` présente un champ: *level*, un entier qui caractérise le niveau de sécurité d'un document et une méthode: *access(u:`User`)*. À ce stade de la modélisation, c'est le document qui gère l'accès. Il reçoit une demande de la part d'un usager u et répond par oui ou non, une fois que le droit d'accès est vérifié. Cette classe permet la création des documents.

- La classe `Clearance` a un seul champ: *codeC*, un entier.

- La classe `Level` a aussi un seul champ: *codeL*, un entier.

Les deux dernières classes servent à l'encodage du tableau 5.1. Pour mieux comprendre, il faut prendre en considération le diagramme d'objets, à la figure 5.4. En général, ce diagramme permet de présenter l'état initial du système. Dans ce cas, il est composé d'un usager `u` et d'un document `d`. De plus, il sert à encoder le tableau de données. Les en-têtes des colonnes sont représentées par des instances de la classe `Clearance` (`enh, confV, secV, topSecV`) et les en-têtes des lignes par des instances de la classe `Level` (`uncl, confL, secL, topSecL`). Les valeurs du tableau sont de type booléen: *Granted* (True) et *Denied* (False). Dans le diagramme de la figure 5.4, une valeur True est représentée par la présence d'une flèche d'un objet `Clearance` vers un objet `Level` et dualement, l'absence de la flèche veut dire False.

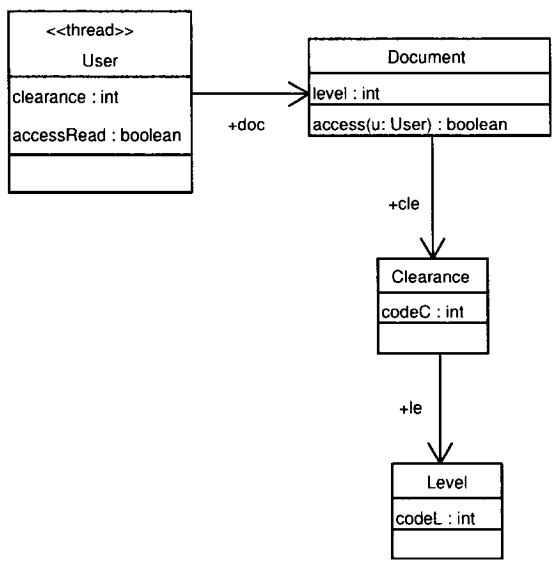

Figure 5.3 Diagramme de classes - Étape 1

Comment naviguer dans un tel diagramme? On se sert des expressions `OCL`.

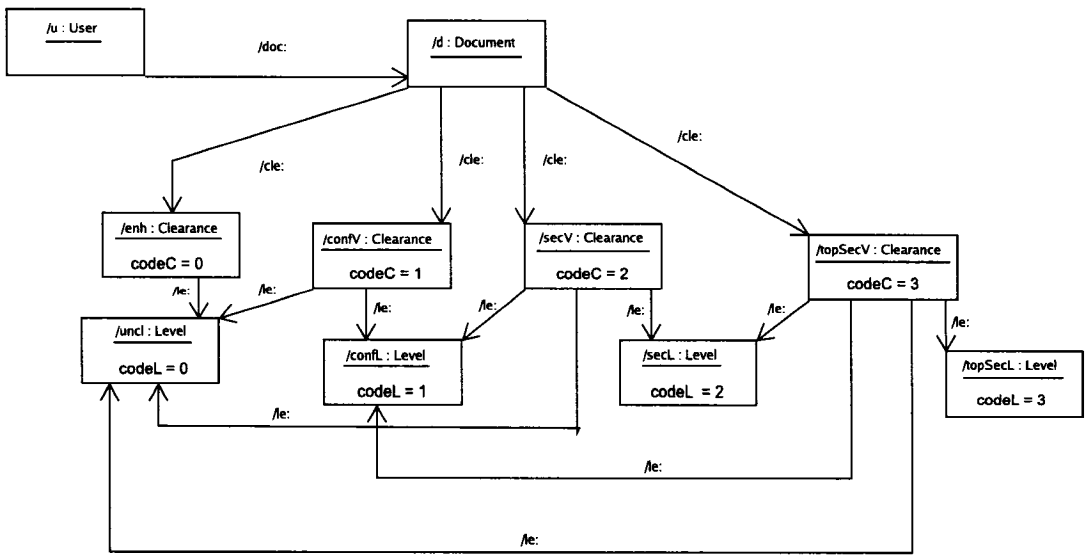

Figure 5.4 Diagramme d'objets - Étape 1

Dans le langage formel d'OCL, l'objet courant est représenté en utilisant le mot clé *self*. On considère que l'objet courant est le document. L'expression OCL *self*.Document.*cle* nous fournit une liste d'objets de type Clearance: [enh, confV, secV, topSecV]. Il est à noter qu'on peut appliquer les fonctions classiques pour opérer sur les collections (figure 4.6, chapitre 4). Le tableau 5.4 présente ainsi plusieurs expressions OCL de navigation et leurs évaluations. Pour chaque cas, une courte explication s'en suit.

Note 5.1 Compte tenu que le type String n'est pas encore utilisable dans la modélisation avec outil SOCLe, les champs de classes normalement de type String seront de type Int. Les tableaux 5.2 et 5.3 présentent les correspondances String - Int pour plusieurs attributs utilisés dans l'étude de cas.

Tableau 5.2 Visibilité, Pays, Rang

Nom	Code
Enhanced	0
Confidential	1
Secret	2
TopSecret	3

Nom	Code
Canada	100
USA	200
UK	300
Germany	400
Egypt	500

Nom	Code
General	99
Officer	88
Civilian	77

Tableau 5.3 Niveau, Cavéat, Projet

Nom	Code
Unclassifield	0
Confidential	1
Secret	2
TopSecret	3

Nom	Code
CEO	10
CANUS	11
CANUSUK	12
NATO	13
UN Coalition	14
USUK	15

Nom	Code
A	1
B	2
C	3
D	4
E	5
F	6

Voici l'interprétation des expressions `OCL` exprimées au tableau 5.4:

1. L'objet courant : `d`.
2. La valeur de l'attribut *cle* (de l'objet courant) est une liste d'objets de type `Clearance`.
3. La liste d'objets de type `Clearance`, pour lesquels l'attribut $codeC = 1$.
4. La valeur de l'attribut *le* de l'objet de type `Clearance` pour lequel l'attribut $codeC = 1$ (`confV`). Cette valeur est une liste d'objets de type `Level`.
5. La valeur de l'attribut *le* pour les objets de type `Clearance` pour lesquels $codeC = 1$ or $codeC = 0$ (`enh`, `confV`). Cette valeur est une liste d'objets de type `Level`. On se retrouve avec la même liste que dans l'exemple 4, donc pas de duplication.
6. Une expression `OCL` booléenne, évaluée à `True`, car dans la liste des objets de type `Level` retrouvée par l'expression 4, il existe un objet pour lequel le champ *codeL* a la valeur 0.

Tableau 5.4 Expressions OCL de navigation

Ex.	Expression OCL	Évaluation
1	*self*	d
2	*self*.Document/*cle*	[enh, confV, secV, topSecV]
3	(*self*.Document/*cle*). filter{x | x.Clearance/*codeC* = 1}	[confV]
4	(*self*.Document/*cle*). filter{x | x.Clearance/*codeC* = 1}. Clearance/*le*	[uncl, confL]
5	(*self*.Document/*cle*). filter{x | x.Clearance/*codeC* = 1∨ x.Clearance/*codeC* = 0}. Clearance/*le*	[uncl, confL]
6	((*self*.Document/*cle*). filter{x | x.Clearance/*codeC* = 0}. Clearance/*le*). exists{y | y.Level/*codeL* = 0}	True

Vérification d'un invariant Le but de cette première partie n'est pas d'effectuer la vérification. On présente quand même un exemple introductif. À ce stade, le système a un comportement simple, et tout ce qu'on peut vérifier est l'encodage du tableau 5.1.

Exemple 5.1 Informellement, on peut exprimer l'invariant suivant: si jamais un usager avec le plus petit niveau de visibilité a accès à un document, alors il est classifié comme **Unclassified** (non protégé).

Formellement, l'invariant prend la forme suivant:

context: User

inv: $(((self.\texttt{User}/accessRead) = \text{True}) \wedge (self.\texttt{User}/clearance = 0)) =>$

$(self.\texttt{User}/doc.\text{forall}\{d \mid d.\texttt{Document}/level = 0\})$

L'invariant est vrai et représente la vérification de la première cellule du tableau 5.1.

Remarque 5.1 Le tableau 5.1 présente une symétrie dans le sens où le nombre de colonnes est égal au nombre de lignes et les valeurs sur la diagonale, et au-dessus sont True. En conséquence, on peut traduire cette contrainte par le garde (5.1), exprimé dans le digramme d'états-transitions correspondant à la classe `Document`. Le diagramme d'objets est réduit à celui de la figure 5.5.

$$\textbf{guard: } u.\texttt{User}/clearance >= d.\texttt{Document}/level \tag{5.1}$$

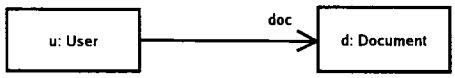

Figure 5.5 Diagramme d'objets réduit - Étape 1

5.3.2 Étape 2 - Tableau Cavéats

Dans cette étape, on considère une extension de l'étude de cas de l'étape 1, en ajoutant des nouvelles données. Cette fois-ci, le système doit respecter les contraintes représentées informellement par les tableaux 5.1 et 5.5.

On utilise l'expérience acquise dans l'étape 1 pour modéliser le tableau 5.5. On a toujours affaire à des valeurs booléennes, mais cette fois-ci, le tableau ne présent pas une symétrie. Le diagramme de classes est représenté dans la figure 5.6. La classe `User` présente un attribut de plus: *country* (un entier) qui représente le pays d'origine d'usager. Pour la classe `Document`, on ajoute l'attribut *caveat*, qui représente le Cavéat du document. Les correspondances String-Int pour les deux nouveaux attributs se retrouvent dans les tableaux 5.2 et 5.3.

Le diagramme d'objets de la figure 5.7 permet de représenter les objets qui pré-

Tableau 5.5 Tableau Caveats

Caveat/Country	Canada	US	UK	Germany	Egypt
CEO	*Granted*	*Denied*	*Denied*	*Denied*	*Denied*
CANUS	*Granted*	*Granted*	*Denied*	*Denied*	*Denied*
CANUSUK	*Granted*	*Granted*	*Granted*	*Denied*	*Denied*
NATO	*Granted*	*Granted*	*Granted*	*Granted*	*Denied*
UN Coalition	*Granted*	*Granted*	*Granted*	*Granted*	*Granted*
USUK	*Denied*	*Granted*	*Granted*	*Denied*	*Denied*

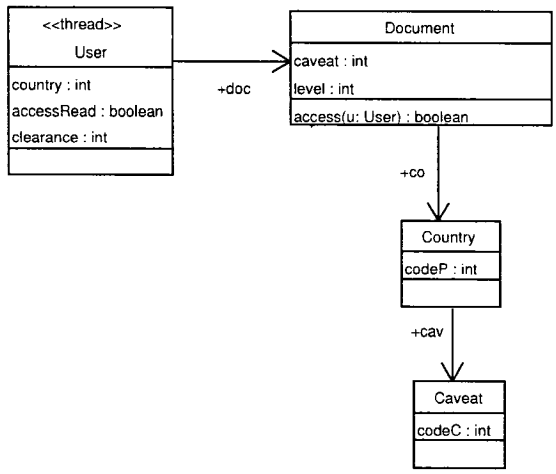

Figure 5.6 Diagramme de classes - Étape 2

sentent un comportement dynamique (qu'on va nommer *objets dynamique*) `u: User`, `d: Document` et aussi les objets qui ne présentent pas un comportement dynamique (qu'on va nommer *objets passifs*) et qui encodent le tableau 5.5. De la même façon qu'au premier tableau, on utilise des instances des deux nouvelles classes: la classe `Country` et la classe `Caveat` pour représenter les en-têtes lignes et les en-têtes colonnes du tableau. La présence d'une flèche d'un objet `Country` vers un objet `Caveat` dénote la valeur booléenne `True`, et l'absence `False`. Les expressions `OCL` facilitent aussi la navigation et la recherche dans un tel tableau. Puisque les valeurs sont quelconques, on ne peut pas appliquer une réduction comme dans le premier tableau. Ce tableau restera tel qu'il est sur la figure 5.7.

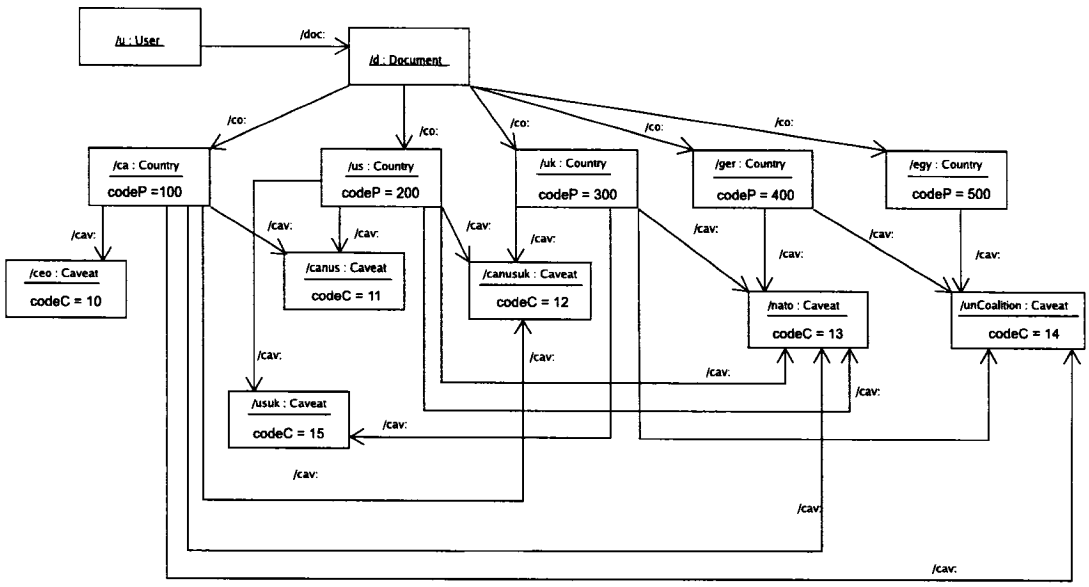

Figure 5.7 Diagramme d'objets - Étape 2

Inconvénients Cette manière de modéliser se montre correcte, bien que dans les deux études de cas (Étape 1 et Étape 2), le système est réduit à un usager et un document. Un document de plus implique une table de plus, donc onze objets dans le diagramme d'objets à ajouter pour chaque document. Cela veut dire que, même si la modélisation s'avère correcte, la complexité de certains diagrammes l'infirme. En plus, il ne faut pas oublier qu'il nous reste encore une table à représenter. L'étape 3 présente succinctement les particularités du tableau 5.6. Les détails de modélisation seront expliqués dans l'étape 4, qui introduit la notion d'héritage.

5.3.3 Étape 3 - Tableau Projet

Dans cette étape, on présente la façon de coder un tableau de valeurs non booléennes, le tableau 5.6. Pour cela, il faut envisager une manière de représenter la valeur de chaque cellule du tableau.

Tableau 5.6 Tableau Projet

Projet/Country	Canada	US	UK	Germany	Egypt
A				*G-O*$^{(1)}$	*G-O-C*
B	*G-O-C*	*G-O-C*			
C	*G*	*G-O*	*G-O-C*	*G-O-C*	
D	*G-O*	*G-O-C*	*G-O-C*		
E	*G-O-C*	*G-O-C*	*G*		
F			*G-O-C*		*G-O*

Légende
G: *Général*
O: Officier
C: Civil

Comment lire l'information dans ce tableau? Exemple: un document faisant partie du projet C peut être lu par des usagers qui ont comme pays d'origine: Canada, US, UK, et Germany. De plus, si l'usager vient du Canada, il doit détenir le rang militaire Général (G).

Le système doit respecter les conditions exprimées dans les trois tableaux: Classification, Caveats et Projet (tableaux 5.1, 5.5 et 5.6).

Pour pouvoir vérifier les droits d'accès en fonction du tableau Projet, on est censé ajouter des attributs dans la classe `User` et la classe `Document`. La classe `User` ajoute l'attribut *rank*, un entier qui représente le rang militaire de l'usager. La classe `Document` ajoute l'attribut *project*, un entier qui représente le projet du document. On utilise toujours des objets pour encoder le tableau. La figure 5.8 présente les objets dynamiques `d1`, `d2` (deux documents) et `u` (un usager), et les objets passifs qui encodent les trois tableaux. L'objet *tabP* et tous les objets en dessous représentent le tableau Projet (des explications additionnelles sont ajoutées pour les trois objets *tabS*, *tabC* et *tabP* dans la section 5.4).

Chaque cellule du tableau est représentée par un objet de type `Cell`, et les trois objets attachés, un de type `Country`, un de type `Projet` et un de type `Rank` donneront respectivement l'information sur le pays, sur le projet et sur le rang militaire correspondants.

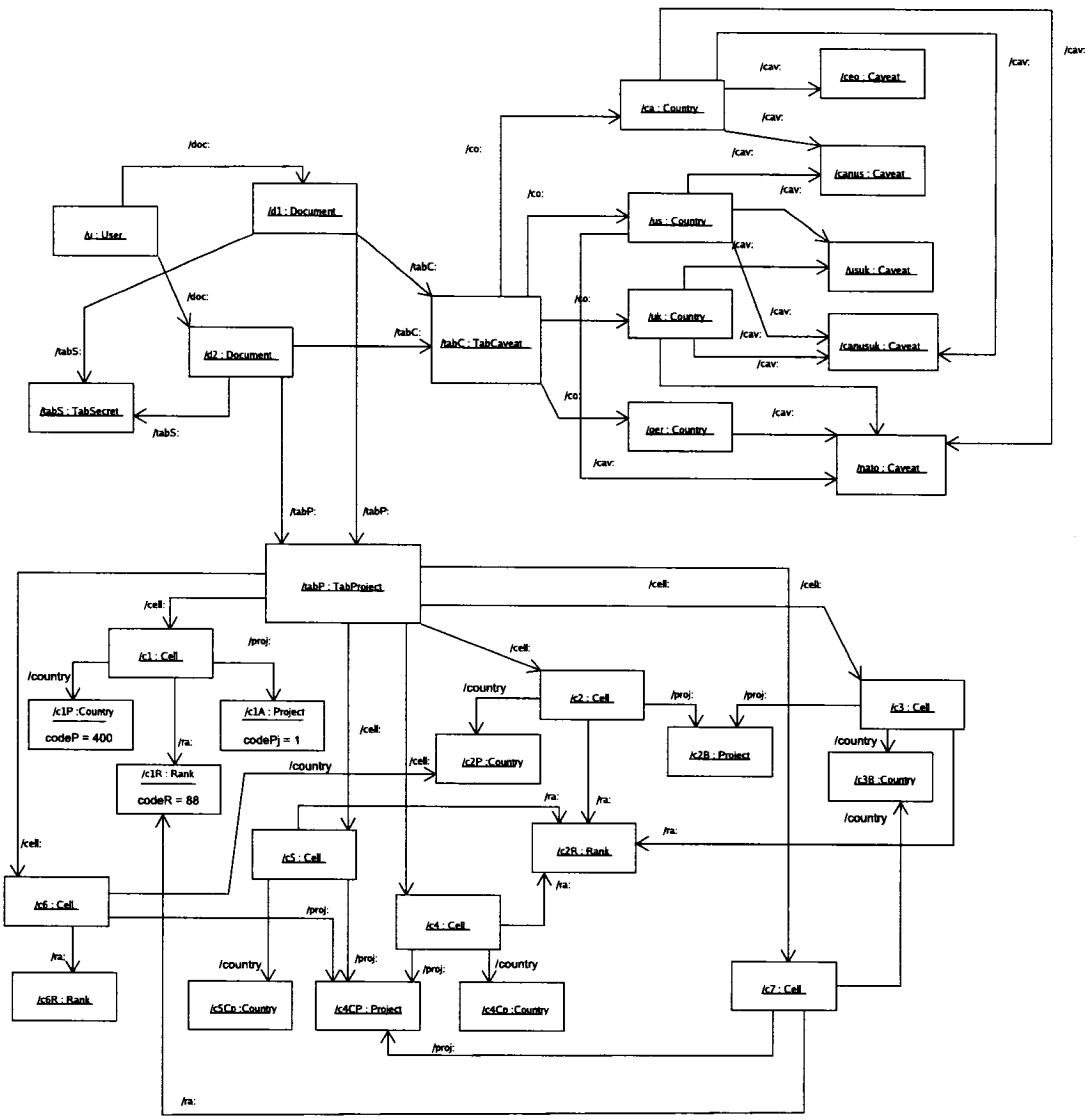

Figure 5.8 Diagramme d'objets - Étapes 3 et 4

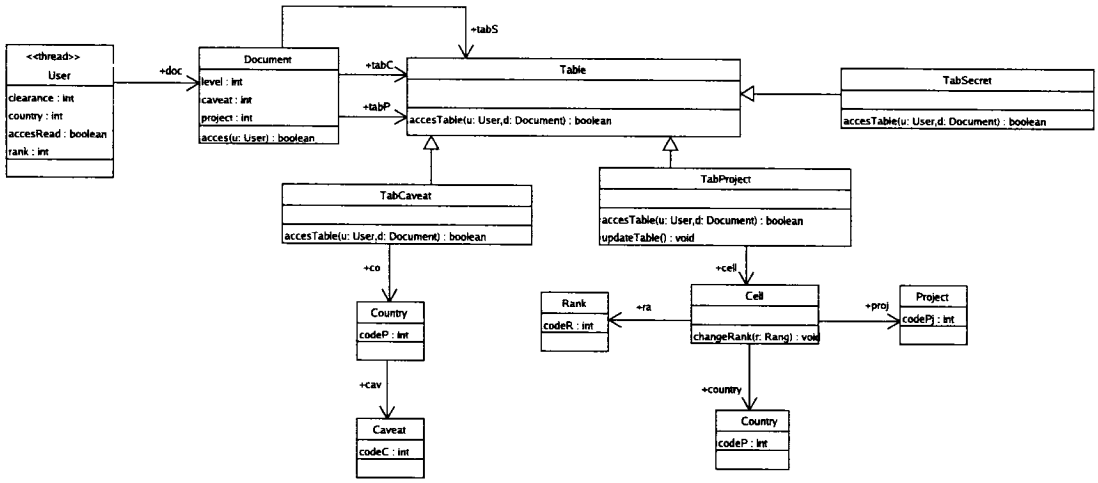

Figure 5.9 Diagramme de classes - Étape 3 et 4

Exemple 5.2 L'objet `c1`, qui réfère à `c1P` (codeP = 400 - Gemany), à `c1R` (codeR = 88 - O-G) et à `c1A` (codePj = 1 - A), veut dire qu'on a représenté la cellule (1) de la table 5.6.

5.3.4 Étape 4 - Héritage

Dans cette étape, on utilise l'héritage. La structure du système est représentée dans le diagramme de classes de la figure 5.9. Une classe `Table` représente une classe abstraite. Elle contient une seule méthode abstraite *accessTable(u:`User`, d:`Document`)*. Cette méthode sera redéfinie dans chaque sous-classe: `TabSecret`, `TabCaveat` et `TabProject`. Elle permet de vérifier l'accès au document `d` par l'usager `u` en fonction de leurs caractéristiques et des contraintes imposées par les tableaux 5.1, 5.5 et 5.6. La technique d'encodage des ces trois tableaux a été présentée dans les étapes précédentes (sous-sections 5.3.1, 5.3.2 et 5.3.3).

5.3.5 Simulation des objets dynamiques

Dans le diagramme d'objets (figure 5.8) on a initialisé un seul usager et un ou deux documents. À l'aide du diagramme d'états-transitions on peut simuler plusieurs usagers et documents. La figure 5.10 présente le diagramme d'états-transitions pour la classe `User`. Le comportement d'usager est simple à ce stade de la modélisation. On ne présente pas en détail le comportement dynamique. L'intérêt de ce diagramme consiste à montrer la façon de modéliser plusieurs usagers.

Chacune des transitions: *u1, u2, u3, u4, u5* modélise une valeur différente pour l'attribut *clearance*. Les transitions *u6, u7, u8, u9, u10* simulent des valeurs différentes pour l'attribut *country* et les transitions *u13, u14, u15* permettent la modélisation de trois valeurs différentes pour l'attribut *rank*. La combinaison de ces valeurs permettront la simulation des différents usagers.

Pour simuler plusieurs documents, on procède de la même manière en utilisant le diagramme d'états-transitions correspondant à la classe `Document`.

Cette façon de modéliser est correcte, mais augmente rapidement le nombre d'états dans le graphe d'exécution; fait dû à l'explosion combinatoire. Pour faciliter la vérification, on a la possibilité de réduire le graphe, en bloquant certaines transition (*guard* = False).

5.3.6 Résultats de vérification

Le tableau 5.7 illustre des résultats de vérifications sur les modèles exposés dans ces trois étapes. Pour chaque modèle, on vérifie un invariant ayant comme contexte la classe `User`. Comme l'invariant est vérifié, pour chaque modèle le graphe d'exécution est calculé au complet. On ne présente pas en détail les spécifications

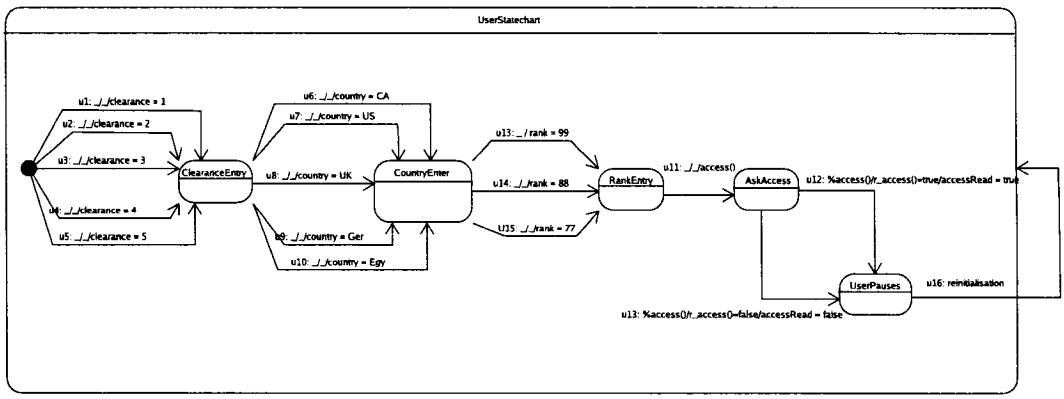

Figure 5.10 Diagramme d'états-transitions User - Étape 4

formelles des invariants.

Tableau 5.7 Résultants pour les étapes 1, 2, 3 et 4

Étape/Résultats	**Étape 1**	**Étape 2**	**Étape 3, 4**
Nb. États	109	1330	3643
Temps (s) OCL^{EXT}/μ-calcul	0.53 / 0.61	43.53 / 48.10	173.38 / 301.09
User	*clearance* : 0, 1, 2, 3	*clearance* : 0, 1, 2 *country* : 100, 200	*clearance* : 0, 1 *country* : 10, 11 *rank* : 77, 88, 99
Document	*level* : (0,1,2,3)	*level* : 0, 1, 2, 3 *caveat* : 10, 11, 12	*level* : 0, 1 *caveat* : 10, 11 *projet* : 1, 2

À chaque étape de la modélisation, le tableau présente le nombre d'états, le temps de calcul et les objets dynamiques simulés (par les attributs et leurs valeurs possibles). Par exemple, à l'étape 2 l'usager est caractérisé par deux attributs: *clearance* et *country*. L'attribut *clearance* peut prendre les valeurs 0, 1 ou 2 et l'attribut *country* peut être soit Canada(100), soit USA(200). Le document présente aussi deux attributs: le niveau de sécurité (*level*), avec les valeurs possibles 0, 1, 2, 3, et le cavéat avec trois valeurs possibles. Le graphe obtenu contient 1330 états. Le temps de calcul avec l'algorithme linéaire OCL^{EXT} est de 43.53 secondes et selon

l'algorithme de μ-calcul est de 48.10 secondes.

On remarque que, pour le même comportement dynamique, un choix de plus dans la plage de valeurs des attributs, implique une explosion de l'espace d'états. Il faut noter aussi que, plus le graphe d'exécution augmente en taille, plus le temps de calcul selon l'approche μ-calcul est plus important comparativement à l'approche OCL^{EXT}. Donc, la vérification à la volée s'avère plus efficace, même pour les cas où on calcule le graphe au complet.

5.4 Modélisation du comportement

Dans cette étape, il faut présenter une modélisation plus complexe. Deux éléments nouveaux sont ajoutés: un contrôleur et le partage des usagers. Cette partie a pour but de modéliser toutes les données d'entrées: les documents, les usagers, les contraintes (les trois tableaux), en s'appuyant sur le comportement dynamique. Celui-ci est décrit par les diagrammes d'états-transitions. La concurrence est prise en compte, causée soit par la présence de plusieurs classes *thread*, soit par la présence d'états composites AND.

5.4.1 Diagramme de classes, diagramme d'objets

Diagramme de classes Le diagramme de classes (figure 5.11), contient toutes les classes déjà détaillées dans les étapes précédentes ainsi que les nouvelles classes `Controller`, `Sender` et `Receiver`.

La classe `Document` comporte trois attributs: le niveau de sécurité (*level*), le cavéat (*caveat*) et le projet (*project*). Les instances de cette classe représentent les documents existants dans les archives.

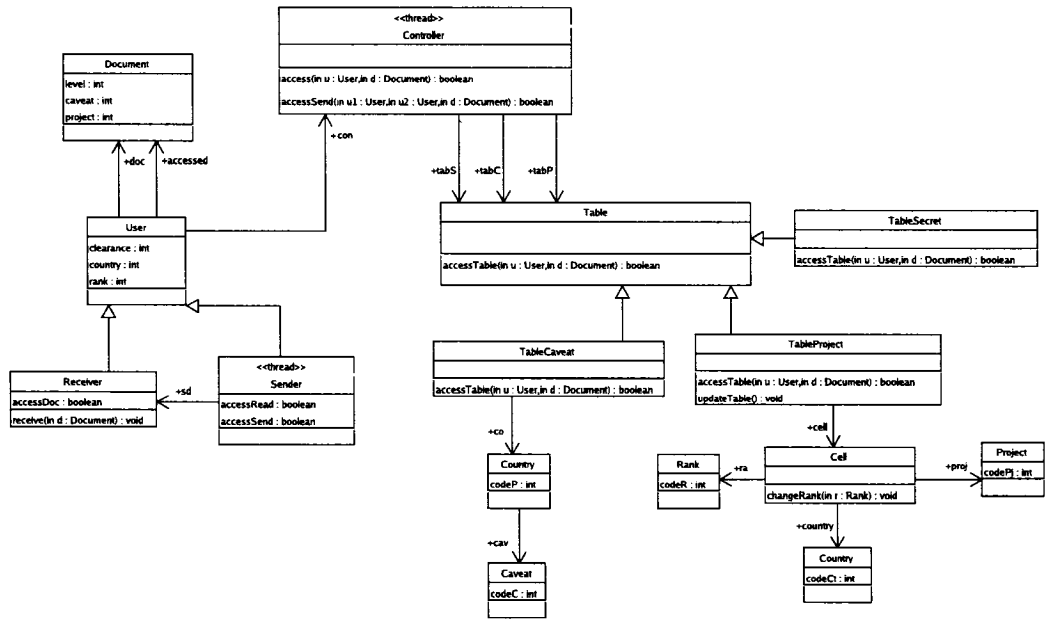

Figure 5.11 Diagramme des classes

La classe `Controller` réfère à un contrôleur de système. Son premier rôle consiste *à contrôler* l'accès (usagers vers documents) et autoriser une communication point-à-point entre deux usagers. Les méthodes correspondantes dans la classe `Controller` sont les suivantes:

- *access(u:User, d:Document)* lorsque l'usager `u` demande l'accès au document `d`.
- *accessSend(u1:User, u2:User, d:Document)* lorsque `u1` demande la permission d'envoyer le document `d` à l'usager `u2`.

Le deuxième rôle du contrôleur est d'*administrer* les trois tableaux de contraintes. Il a donc le pouvoir de changer les droits d'accès. Le tableau Classification (tableau 5.1) ne peut pas être modifié, car il représente une contrainte stricte. Par contre, les tableaux Caveats et Projet peuvent être modifiés. Dans cette étude de cas on

effectue la modélisation avec la possibilité de changer le tableau Projet. C'est le contrôleur qui peut effectuer ce changement, en appelant la méthode *updateTable()* de la classe `TableProject`. Les détails du comportement sont exposés dans la section 5.4.2.

La classe `User` comporte trois attributs: le niveau de visibilité (*clearance*), le pays d'origine (*country*) et le rang militaire (*rank*). On se servira encore une fois de l'héritage pour modéliser deux types d'usagers: un de type `Sender` et un de type `Receiver`. L'usager de type `Sender` présente deux comportements. Il peut essayer/accéder aux documents ou il peut essayer/envoyer des documents à un autre usager de type `Receiver`. La classe `Sender` et la classe `Receiver` sont dérivées de la classe `User`.

Les variables booléennes *accessRead*, *accessSend*, *accessDoc* sont utilisées comme indicateurs d'accessibilité aux documents. Elles indiquent respectivement si un `Sender` a le droit de lire ou d'envoyer un document, et si un `Receiver` a le droit de lire un document. Elles donnent plus d'expressivité aux contraintes surtout si on veut faire une vérification sur un objet précis.

Les attributs *doc*, *accessed* et *sd* représentent la liste de documents existants dans les archives, la liste de documents auxquels les usagers ont accès et respectivement la liste de `Receivers` avec lesquels les `Senders` sont en contact.

La classe `Receiver` qui contient la méthode *receive(d: Document)*, sert au receveur pour demander un document à l'envoyeur.

Les classes qui encodent les tableaux 5.1, 5.5 et 5.6, restent les mêmes, à l'exception de la classe `TableProject` et la classe `Cell`, qui changent, puisqu'elle utilisent respectivement la méthode *updateTable()* et *changeRank(r:Rank)*.

Diagramme d'objets Le diagramme d'objets (figure 5.12), contient d'un côté les objets passifs, qui codent les trois tableaux:

- l'objet `tabS`, de type `TableSecret` qui représente le tableau 5.1.
- l'objet `tabC`, de type `TableCaveats`, avec les objets attachés (de type `Country` et de type `Caveat`), qui représente le tableau 5.5.
- l'objet `tabP`, de type `TableProject`, avec les objets attachés (de type `Cell`, de type `Rank`, de type `Country`, et de type `Project`), qui représente le tableau 5.6.

Il contient aussi les objets dynamiques:

- l'objet `c`, de type `Controller`, qui représente le contrôleur du système.
- l'objet `d`, de type `Document`,
- l'objet `uS`, de type `Sender`
- l'objet `uR`, de type `Receiver`

Note 5.2 Pour modéliser les usagers et les documents, on se sert du diagramme d'objets. Il suffit de changer les valeurs d'attributs d'objets dans ce diagramme, pour simuler d'autres objets. On élimine le non-déterminisme utilisé dans les étapes précédentes dans le but d'éviter l'explosion combinatoire. Dans ce stade de modélisation, on s'intéresse plutôt au comportement dynamique et à la rechercher d'erreurs de conception.

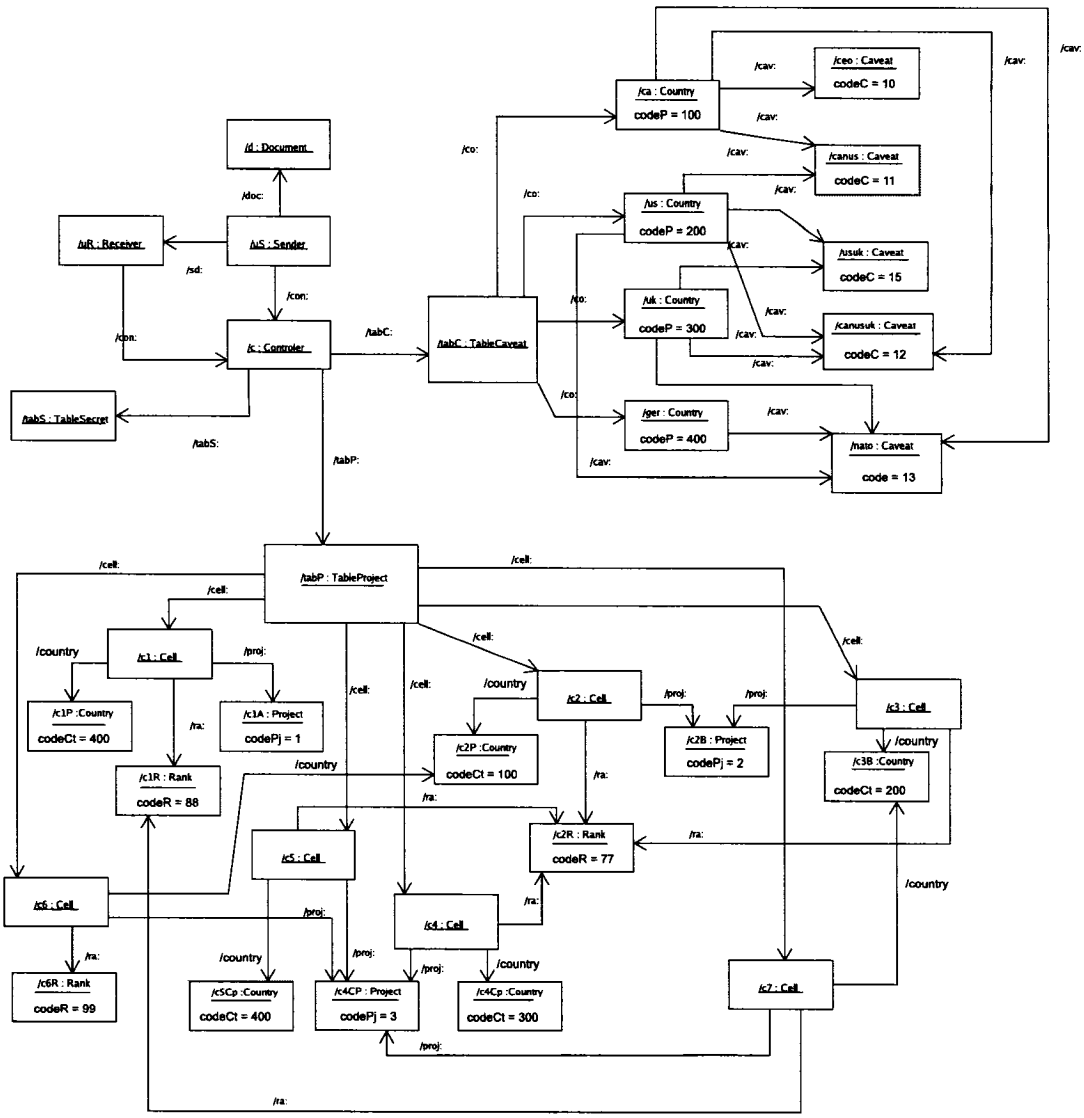

Figure 5.12 Diagramme d'objets

5.4.2 Diagrammes d'états-transitions

Les diagrammes d'états-transitions décrivent le comportement dynamique du système. À chaque classe correspond un diagramme d'états-transitions. On se rappelle qu'un tel diagramme comporte minimum un état initial hébergé dans un état composé (type OR). La classe qui présente un tel diagramme minimal, n'a pas vraiment un comportement dynamique. Son rôle est plutôt statique. C'est le cas d'encodage de tableaux, expliqué dans la première partie. Cette section présentera les diagrammes d'états-transitions pour les classes qui ont un comportent dynamique.

5.4.2.1 Diagramme d'états-transitions `Sender`

Le diagramme qui se trouve à la figure 5.13 modélise le comportement dynamique de la classe **`Sender`**. Il est composé de deux états de type OR: *AccessRead* et *AccessSend* qui vont être en concurrence.

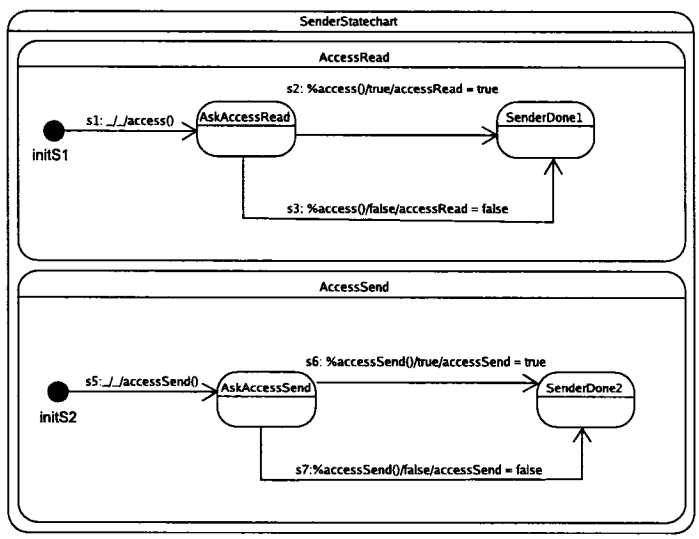

Figure 5.13 Diagramme d'états-transitions - **`Sender`**

Diagramme d'états-transitions `Sender` **-** ***AccessRead*** Un objet de la classe `Sender` demande au contrôleur, d'accéder à un objet de la classe `Document`. Cette demande se matérialise par un appel (une action) de la méthode *access(u:*`User`, *d* :`Document`). La méthode appartient à la classe `Controller` et elle exige deux paramètres: un usager (type `User`) et un document (type `Document`).

L'accès se fait par une expression `OCL`, à partir de l'objet demandeur. Dans l'appel de méthode, il faut bien spécifier les paramètres. Ici, l'usager sera l'objet courant *self* et le document doit être choisi dans la liste de documents existants. On prend le premier de la liste *head(self.*`User`*/doc)*. Dans ce cas-ci, c'est le seul élément de la liste (voir diagramme d'objets, figure 5.12).

Une fois la demande envoyée et donc la transition *s1* tirée, l'objet de type `Sender` se met en attente (état: *AskAccessRead* - activé, état initial *initS1* - désactivé). La transition *s1* modélise aussi les re-initialiserions des deux champs: *accessRead* = False et *accessed* = null (voir les transitions formelles au tableau 5.8).

La prochaine transition (*s2* ou *s3*) sera déclenchée par le retour de la méthode. Si la réponse est positive, l'objet demandeur effectuera la transition *s2* qui aura comme effet de mettre à jour sa variable booléenne *accessRead* =True et d'enregistrer le document dans la liste d'accès (`User`*/accessed = self.*`User`*/doc*) (Il est à noter qu'il y a un seul document dans la liste *doc*). Sinon, la transition qui est tirée sera *s3*.

L'état *SenderDone* modélise une de ces deux possibilités: l'usager est en possession du document ou l'usager reçoit un refus de la part du contrôleur. Cet état est activé et l'état *AskAccessRead* est désactivé. On considère qu'après un certain temps l'usager n'a plus l'accès au document et qu'il le dépose dans les archives. Un nouvel accès suppose une nouvelle demande.

Tableau 5.8 Transitions Sender - *AccessRead*

s1	
trigger :	$\star$
guard :	True
action :	Sender/*accessRead* = False; User/*accessed* = null; *self*.User/*con*.Controller/*access*(*u* :User, *d* : Document) : Bool[*self*, *head*(*self*.User/*doc*)]
s2	
trigger :	Controller%*access*(*u* :User, *d* :Document) :Bool
guard :	*r_access* = True
action :	Sender/*accessRead* = True; User/*accessed* = *self*.User/*doc*
s3	
trigger :	Controller%*access*(*u* :User, *d* :Document) :Bool
guard :	*r_access* = false
action :	-

Diagramme états-transitions Sender - ***AccessSend*** Presque le même scénario se produit quand l'usager de type Sender demande le droit de communiquer avec un usager Receiver. Cette fois-ci, l'objet courant appelle la méthode *accessSend(u1:*User, *u*2 :User, *d* :Document) du Controller. Les paramètres qui sont envoyés sont:

- u1 = *self*, l'objet courant.
- u2 = *head(self.*Sender*/sd)*, le premier élément de la liste de receveurs avec lequel l'objet courant est en contact. Dans ce cas-ci, on a qu'un seul receveur.
- d = *head(self.*User*/doc)*, le document qui est le premier élément de la liste d'objets Document.

Note 5.3 Il faut remarquer que les attributs *doc* et *sd* sont de type liste, donc il faut bien spécifier l'élément de la liste qui nous intéresse.

Les transitions au complet sont présentées au tableau II.2, annexe II.2.

5.4.2.2 Diagramme états-transitions `Controller`

Ce diagramme (figure 5.14) est composé de trois états de type OR: *AskAccess*, *AskSend* et *ChangeTable* qui vont se retrouver en concurrence.

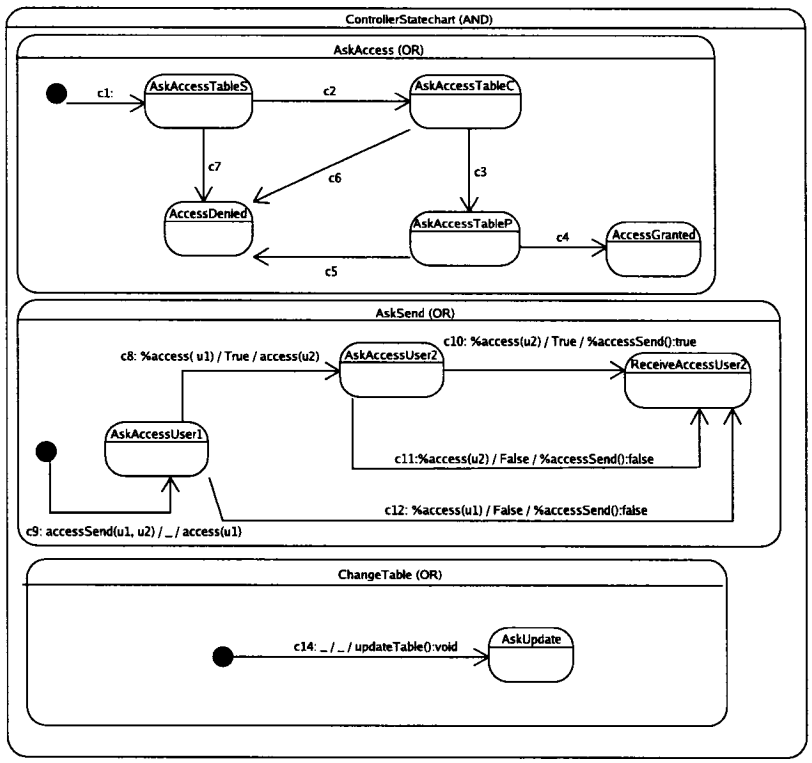

Figure 5.14 Diagramme d'états-transitions - `Controller`

Diagramme états-transitions `Controller` - *AskAccess* Le `Controller` reçoit l'appel de méthode *access(u:*`User`, d : `Document`). À son tour, il fait appel à la méthode *accessTab(u:*`User`, d :`Document`) de la classe `Table`. Cette classe détient la signature de la méthode qui vérifie les droits d'accès. Cette méthode

est redéfinie dans chaque classe dérivée. La présence des trois pointeurs: *tabS*, *tabC*, *tabP* permet l'accès direct à la table désirée. La vérification est faite successivement en appelant les tables `TableSecret`, `TableCaveat` et `TableProject` l'une après l'autre. Si tous les droits sont respectés, la table répond positivement au `Controller`. Si un des droits est violé, la réponse est négative. Informellement, ce scénario se présente comme suit:

c1: Le `Controller` reçoit la demande d'accès et envoie à son tour une demande de vérification à la première table, `TableSecret`.
L'état *AskAccessTableS* est activé et l'état initial *initC1* est désactivé.

c7: Réponse négative, le `Controller` répond par un refus à l'usager.
L'état *AccessDenied* est activé et l'état *AskAccessTableS* est désactivé.

c2: Réponse positive, le `Controller` envoie une demande à la `TableCaveat`.
L'état *AskAccessTableC* est activé et l'état *AskAccessTableS* est désactivé

c6: Pas de droits d'accès.
L'état *AccessDenied* est activé et *AskAccessTableC* est désactivé.

c3: Réponse positive, le `Controller` interroge la table `TableProject`.
L'état *AskAccessTableP* est activé et *AskAccessTableC* est désactivé.

c5: Pas de droits d'accès.
L'état *AccessDenied* est activé et *AskAccessTableP* est désactivé.

c4: Réponse positive, le `Controller` permet l'accès.
L'état *AccessGranted* est activé et l'état *AskAccessTableP* est désactivé.

Formellement, les transitions sont présentées dans le tableau II.3, annexe II.2.

Diagramme d'états-transitions `Controller` - *AskSend* D'une manière informelle, les transitions qui composent ce diagramme sont:

c9: Le `Controller` reçoit une demande de communication point-à-point.
Le `Controller` envoie une demande de vérification d'accès pour l'usager demandeur (*u1*).

c8: Le `Controller` reçoit le droit d'accès (True) pour l'usager `u1`.
Le `Controller` envoie la demande de vérification d'accès pour le receveur (`u2`).

c12: Le `Controller` reçoit le droit d'accès (False) pour l'usager `u1`.
Le `Controller` envoie un refus de permission.

c10: Le `Controller` reçoit le droit d'accès (True) pour l'usager `u2`.
Le `Controller` envoie une réponse positive à l'objet demandeur.

c11: Le `Controller` reçoit le droit d'accès (False) pour l'usager `u2`.
Le `Controller` envoie une réponse positive à l'objet demandeur.

Formellement, les transitions sont présentées au tableau II.4, annexe II.2.

Diagramme d'états-transitions `Controller` - *ChangeTable* L'administrateur (ici le `Controller`) prend la décision de changer certaines données dans les tableaux. Dans ce cas-ci il change l'information dans le tableau Projet. Le changement se fait à l'aide de la méthode *updateTable()* de la classe `TableProjet`. L'appel de cette méthode est matérialisé par la seule action de la transition *c14*, formalisée dans le tableau II.4, annexe II.2.

5.4.2.3 Diagramme d'états-transitions `Table`

Le comportement dynamique de la classe `Table` est réduit à un état initial (figure 5.15). La redéfinition de la méthode *accessTable()* dans les classes dérivées implique un comportement dynamique délégué.

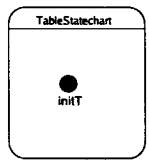

Figure 5.15 Diagramme états-transitions - `Table`

5.4.2.4 Diagramme d'états-transitions `TableSecret`

Ce diagramme (figure 5.16) modélise le comportement dynamique de la classe `TableSecret` qui dérive de la classe `Table`. La table reçoit une demande de vérification de la part du `Controller`, vérifie les droits d'accès (cohérence entre la visibilité de l'usager et niveau de sécurité du document) et donne une réponse. Deux transitions traduisent ce comportement:

- *ts1*: Demande d'accès, garde vrai, réponse positive.
- *ts2*: Demande d'accès, garde faux, réponse négative

La construction du garde appartenant à ces deux transitions a déjà été détaillée dans sous-section 5.3.1 et formellement on les retrouve au tableau II.5, annexe II.2.

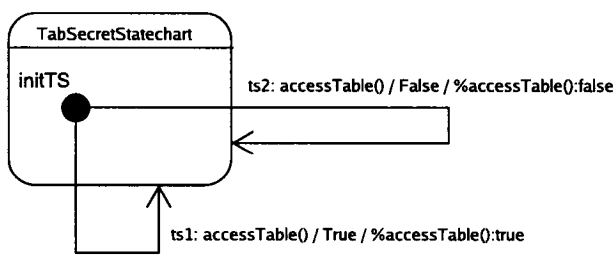

Figure 5.16 Diagramme états-transitions - `TableSecret`

5.4.2.5 Diagramme d'états-transitions `TableCaveat`

Ce diagramme (figure 5.17) modélise le comportement dynamique de la classe `TableCaveat`. Un appel de la méthode *accesTable(u:*`User`, *d* :`Document`) réveille une des deux transitions existantes. La table passe à la vérification des droits et envoie une réponse. Donc, le comportement est simple matérialisé par les deux transitions *tc1* et *tc2*. La difficulté dans ce diagramme est dû à la complexité d'expression `OCL` dans le garde de chaque transition. Le garde vérifie le droit d'accès exprimé par le tableau Caveat (cohérence entre le pays d'origine de l'usager et le cavéat du document). La logique de l'expression `OCL` pour ce garde est donnée au tableau 5.9. On se souvient qu'on utilise le diagramme des objets de la figure 5.12. La formalisation complète de ces transitions se retrouve au tableau II.6, annexe II.2.

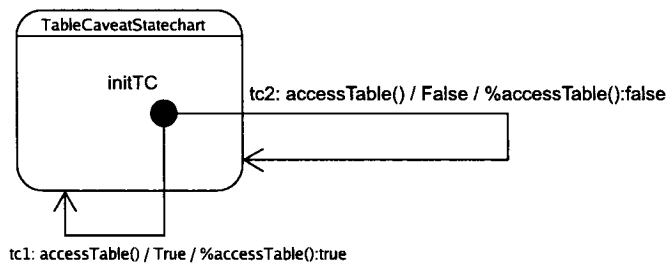

Figure 5.17 Diagramme états-transitions - `TableCaveat`

Pour mieux comprendre, il faut prendre un exemple. Soit un usager `uS` (décrit au tableau 5.10) qui essaie d'accéder au document `d`. L'évaluation de l'expression `OCL` du garde correspondant, est détaillée dans le tableau 5.11.

Tableau 5.9 Garde d'accès Table Caveats

Formule	Expression OCL	Évaluation
ϕ_1:	$self$	l'objet courant
ϕ_2:	ϕ_1.TabCaveat/co	l_1: liste d'objets Country
ϕ_3 :	$\phi_2.filter\{x \mid x$.Country/$codeP = u$.User/$country\}$	l_2: l_1 filtré en fonction du pays d'origine de l'usager qui veut y accéder
ϕ_4 :	ϕ_3.Country/cav	l_3: liste d'objets Caveat correspondante à la liste l_2
guard:	$\phi 4.exists\{y \mid y$.Caveat/$codeC = d$.Document/$caveat\}$	True or False s'il existe un objet dans la liste l_3 dont le code égal au *Cavéat* du document

Tableau 5.10 Objets: uS et d

	$clearance = 1$		$level = 1$
uS:	$country = 100\ (Canada)$	**d:**	$caveat = 10\ (CEO)$
	$rank = 99\ (G - O - C)$		$projet = 2\ (B)$

5.4.2.6 Diagramme d'états-transitions TableProject

Ce diagramme (figure 5.18) modélise le comportement dynamique de la classe TableProject qui est aussi dérivée de la classe Table. Une transition de plus permet de modéliser la demande de l'administrateur et de changer certaines informations dans le tableau Projet. Plus précisément, l'action de la transition *tp3* est une demande pour la classe Cell d'effectuer le changement en lui envoyant comme paramètre le premier objet de type Rank existant dans la liste *ra*. La construction du garde utilisé sur les transitions *tp1* et *tp2* est expliquée ci-dessous:

- ϕ_1: $self$ est l'objet courant
- ϕ_2: ϕ_1.TableProject/$cell$ est la liste d'objets de type Cell

Soit x une cellule.

Tableau 5.11 Garde d'accès Table Caveats - Exemple

Formule	Évaluation
ϕ_1 :	tabC
ϕ_2 :	[ca, us, uk, ger]
ϕ_3 :	[ca]
ϕ_4 :	[ceo, canus, causuk, nato]
guard:	True

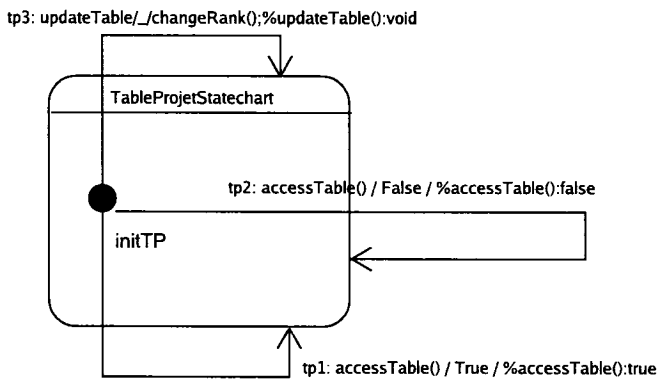

Figure 5.18 Diagramme états-transitions - `TableProject`

x.`Cell`/*country* = *listePays* est une liste d'objets `Country` correspondant à la cellule x

(x.`Cell`/*country*).1 = *paysX* est le premier élément dans la *listePays*, donc un objet de type `Country`

((x.`Cell`/*country*).1).`Country`/*codeCt* est la valeur du champ *codeCt* correspondant au pays *paysX*

- ϕ_3: $\phi_2.filter\{x \mid ((x.\texttt{Cell}/country).1).\texttt{Country}/codeCt = u.\texttt{User}/country\}$ est la liste d'objets `Cell` filtrée avec la condition que le *codeCt* correspondant au pays de la cellule soit égal au code pays de l'usager.

- ϕ_4:ϕ_3.$filter\{y \mid ((y.\texttt{Cell}/proj).1).\texttt{Project}/codePj = d.\texttt{Document}/project\}$ est la liste ϕ_3 des Cell, filtrée selon la condition que le code du projet de la cellule est égal à la valeur du champ projet du Document.

- **guard:** $\phi_4.exists\{z \mid ((z.\texttt{Cell}/ra).1).\texttt{Rank}/codeR <= u.\texttt{User}/rank\}$ est ici finalement une seule cellule pour laquelle le son code militaire est plus petit ou égal à la valeur du champ *rank* de l'usager.

Pour illustrer l'évaluation d'un tel garde, il faut considérer l'exemple précèdent (tableau 5.10). Les expressions `OCL` et leurs évaluations se retrouvent dans le tableau 5.12.

Tableau 5.12 Garde d'accès Table Projet - Exemple

Formule	**Expression `OCL`**	**Évaluation**
ϕ_1 :	$self$	tabP
ϕ_2 :	ϕ_1.`TableProject`/$cell$	[c1, c2, c3, c4, c5, c6, c7]
ϕ_3 :	$\phi_2.filter\{x \mid ((x.\texttt{Cell}/country).1).$ `Country`/$codeCt = u.\texttt{User}/country\}$	[c2, c6]
ϕ_4 :	$\phi_4.filter\{y \mid ((y.\texttt{Cell}/proj).1).$ `Project`/$codePj = d.\texttt{Document}/project\}$	[c2]
guard:	$\phi 4.exists\{z \mid ((z.\texttt{Cell}/ra).1).$ `Rank`/$codeR <= u.\texttt{User}/rank\}$	True

5.4.2.7 Diagramme d'états-transitions `Cell`

Ce diagramme d'états-transitions (figure 5.19), correspond à la classe `Cell`. Un état initial, une transition et un état final composent ce diagramme. La cellule reçoit la demande de changement de la part de la table `Project` et remplace la liste d'objets `Rank` existants par une nouvelle liste construite avec l'objet qu'il reçoit en paramètre: r. Formellement, la transition est présentée au tableau 5.13.

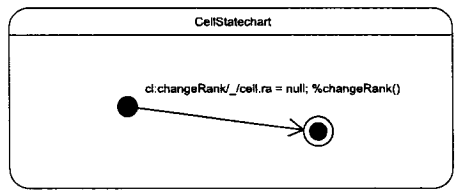

Figure 5.19 Diagramme états-transitions - `Cell`

Tableau 5.13 Transitions `Cell`

cl	
trigger :	`Cell`/*changeRank*(*r* :`Rang`) :void
guard :	True
action :	`Cell`/*ra* = *r* ::null ; `User`/*accessed* = *d* ::null; `Cell`%*changeRank*(*r* :`Rang`)

5.4.2.8 Diagramme d'états-transitions `Receiver`

Ce diagramme d'états-transitions correspond à la classe `Receiver`. Une seule transition le compose. Si cette transition est tirée le `Sender` envoie un document au `Receiver`. La représentation graphique est dans la figure 5.20 et la transition formelle se retrouve au tableau 5.14.

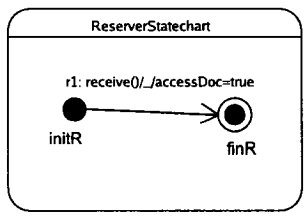

Figure 5.20 Diagramme états-transitions - `Receiver`

Tableau 5.14 Transitions `Receiver`

r1	
trigger :	`Receiver`/*receive*(*d* :`Document`) : *void*
guard :	True
action :	`Receiver`/*accesDoc* = True; `Receiver`^*receive*(*d* :`Document`)

5.5 Vérification des contraintes

5.5.1 Vérification des droits d'accès

On utilise les invariants pour vérifier les droits d'accès, donc la bonne modélisation des trois tableaux (Classification, Caveats, Projet). Pour cette partie, on suppose que le contrôleur joue son premier rôle qui consiste à vérifier les demandes d'accès.

Pour s'assurer que le contrôleur ne change pas l'information dans la table `Project`, on bloque la transition *c14*, dans le diagramme d'états-transitions du contrôleur. Si le tableau est correctement encodée l'invariant doit être vérifié.

Tableau Classification - inv I_1 Les conditions d'accès imposées par le tableau 5.1 peuvent être vérifiés par un invariant. Informellement, un usager a le droit d'accès aux documents qui ont un niveau de sécurité égal ou plus petit que son niveau de visibilité. Le contexte de l'invariant est la classe `Sender`, mais il peut être exprimé dans le contexte de la classe `User` ou la classe `Receiver`. Formellement, il est exprimé à l'aide du patron suivant.

context: `Sender`

inv: (*self*.`User`/*accessed*).forall{*d*|*self*.`User`/*clearance* >= *d*.`Document`/*level*}

On se souvient que *accessed* représente la liste de documents accédé par les usagers.

Tableau Caveat - inv I_2 Cet invariant vérifie les conditions imposées par le tableau 5.5, qui exprime la liaison entre le pays d'origine de l'usager et le cavéat du document.

contexte: Sender

inv: $(self.\texttt{User}/accessed).\text{forall}\{d \mid ((((self.\texttt{User}/con).\texttt{Controller}/tabC).$
$\texttt{TabCaveat}/co).$
$\text{filter}\{x \mid x.\texttt{Country}/codeP = self.\texttt{User}/country\}.\texttt{Country}/cav.$
$\text{exists}\{y \mid y.\texttt{Caveat}/codeC = d.\texttt{Document}/caveat\})\}$

Tableau Projet - inv I_3 Cet invariant vérifie les conditions imposées par la tableau 5.6, qui exprime la liaison entre le projet du document, le pays d'origine et le rang militaire de l'usager.

contexte: Sender

inv: $(self.\texttt{User}/accessed).\text{forall}\{d \mid ((((((self.\texttt{User}/con).\texttt{Controller}/tabP).$
$\texttt{TableProject}/cell).$
$\text{filter}\{x \mid ((x.\texttt{Cell}/country).1).\texttt{Country}/codeCt = self.\texttt{User}/country\}).$
$\text{filter}\{y \mid ((y.\texttt{Cell}/proj).1).\texttt{Project}/codePj = d.\texttt{Document}/project\}).$
$\text{exists}\{z \mid ((z.\texttt{Cell}/ra).1).\texttt{Rank}/codeR <= self.\texttt{User}/rank\})\}$

Tableaux Classification, Caveat, Projet - inv I_4 Si jamais un usager a accès à un document, alors il respecte la politique d'accès exprimée par les trois tableaux

5.1, 5.5 et 5.6.

contexte: `Sender`

inv: $I_1 \wedge I_2 \wedge I_3$

Tableau 5.15 Invariants - Conditions d'accès

Inv - Résultats / **Tech.**	I_1 Nb. états	True Temps	I_2 Nb. états	True Temps
μ-calculus	88	1.14	88	1.29
OCL^{EXT} - naïf	88	1.10	88	1.25
OCL^{EXT} - linéaire	88	1.09	88	1.24
Inv - Résultats / **Tech.**	I_3 Nb. états	True Temps	I_4 Nb. états	True Temps
μ-calculus	88	1.45	88	1.65
OCL^{EXT} - naïf	88	1.41	88	1.63
OCL^{EXT} - linéaire	88	1.35	88	1.52

Dans ce cas, la concurrence est réduite, car il y a qu'une seule pile d'appels (*thread*) en fonction dans la classe `Sender`. La concurrence dans les digrammes d'états-transitions en est même réduite (fait à la transition *c14* bloquée).

Résultat d'une concurrence faible et d'un petit nombre d'objets actifs (un `Sender`, un `Receiver` et un `Document`), le système n'a pas un comportement dynamique compliqué. Le graphe d'exécution est petit (88 états) et consiste en une séquence normale: demande d'accès, vérification des droits d'accès et acceptation (ou refus). L'invariant est vérifié et donc le graphe d'exécution est calculé au complet, pour les deux algorithmes de vérification adoptés: le μ-calcul et OCL^{EXT} à la volée. Puisque la contrainte est vérifiée, on est assuré que la modélisation des tableaux (conditions d'accès) est correcte.

Pour ce cas, la vérification à la volée ne se montre guère plus avantageuse que la technique traditionnelle (μ-calcul). C'est le cas des graphes de petites tailles

et d'invariant vérifié. Pour des graphe de plus grande taille, d'autres études de cas (voir l'annexe II.2, section II.2) ont montré que la technique à la volée est beaucoup plus efficace et ce, même si on est obligé de construire complètement le graphe. Dans le cas où la contrainte est falsifiée, il y a un gain important en temps et en ressources.

5.5.2 Analyse d'erreur

Dans l'idée de continuer le développement de l'étude de cas, on se doit d'ajouter plus de comportements dans le système.

On se souvient que le tableau Classification est une contrainte stricte, elle ne peut pas être modifiée. Par contre, les deux autres tableaux: Caveats et Projet peuvent être modifiés. En débloquant la transition *c14* dans le diagramme d'états-transition du `Controller`, une mise à jour de la table `Project` est possible. On vérifie l'invariant I_4 pour ce nouveau comportement. Dans ce cas, le contexte pour l'invariant est la classe `Receiver`. Ce choix est motivé du fait qu'on essaie de mettre en valeur l'efficacité de la technique à la volée. Si on garde la classe `Sender` comme contexte pour l'invariant, la vérification est effectuée en premier sur l'objet de type `Sender` et, après, sur l'objet de type `Receiver`. Après la mise à jour de la table `Project` (le tableau Projet), l'objet de type `Sender` garde toujours le droit d'accès. Ce qui implique que l'invariant de cet objet est vérifié et qu'on est obligé de construire au complet le graphe. Si la contrainte est exprimée dans la classe `Receiver`, la vérification doit commencer par l'objet de ce type.

contexte: `Receiver`

inv: I_4

Tableau 5.16 Invariant falsifié

Tech/Résultats	I_4	Nb. États	Temps (s)
μ-calculus	False	941	1408.4
OCL^{EXT} naïf	False	244	5.15
OCL^{EXT} linéaire	False	244	9.47

Les résultats exprimés au tableau 5.16, obtenus en appliquant les deux techniques de vérification, démontrent que l'invariant est falsifié. Pourtant on a vu dans les vérifications précédentes que la politique d'accès est bien modélisée. L'erreur est causée par la concurrence du système qui est présente sous deux formes: la présence de deux *threads*, sur la classe `Sender` et `Controller`, et des états composés qui travaillent en concurrence dans les digrammes d'états-transitions du `Controller`. L'exemple schématisé à la figure 5.21 illustre comment la concurrence mal gérée permet une fuite d'information. Le scénario est le suivant:

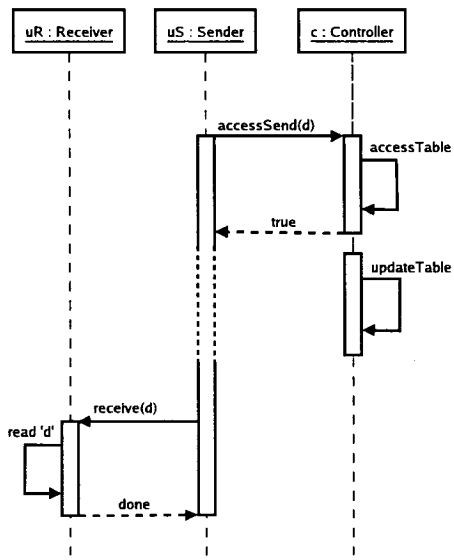

Figure 5.21 Diagramme de séquences - Erreur

- L'usager `uS` demande la communication point à point avec l'usager `uR` dans le but d'envoyer le document *d*.

- Le contrôleur `c` vérifie l'accès pour `uS`.
- Le contrôleur `c` vérifie l'accès pour `uR`.
- Le contrôleur `c` permet la communication point à point.
- L'administrateur change l'information dans la table `Project`, et par la suite l'usager `uR` perd le droit d'accès.
- Les deux usagers détiennent le droit de communication et donc l'usager `uR` accède au document.

Cette contrainte violée démontre comment l'outil SOCLe permet de découvrir des erreurs lors de la phase de conception. Dans ce cas-là l'erreur est causée par la concurrence dans le diagramme d'états-transitions. On se rend du compte de la mauvaise modélisation du *statechart* du contrôleur. De plus la technique à la volée se montre très efficace. Après 244 états et un temps de 5.15s (le cas de l'algorithme naïv), la réponse est donnée.

5.5.3 Un objet de plus

Dans ce paragraphe, on remarque qu'une explosion combinatoire est causée par l'ajout d'un seul objet. Le comportement dynamique du système reste le même. L'objet ajouté est une instance d'une classe *thread*. Le seul diagramme qui change est le diagramme d'objets (figure 5.22). On vérifie l'invariant I_4. Avec la technique à la volée, l'outil répond après 3185 états. Par contre, en appliquant le μ-calcul, l'outil arrive à calculer plus de 10000 états, mais le graphe n'est pas construit au complet. Donc pas de réponse pour la vérification. Dans de tel cas, la vérification à la volée se montre très efficace.

Tableau 5.17 Résultants d'invariant - trois usagers

Tech/Résultats	I_4	Nb. États	Temps (s)
μ−calculus	False	10000 (pas encore fini)	N/A
OCL^{EXT} linéaire	False	3185	193.66

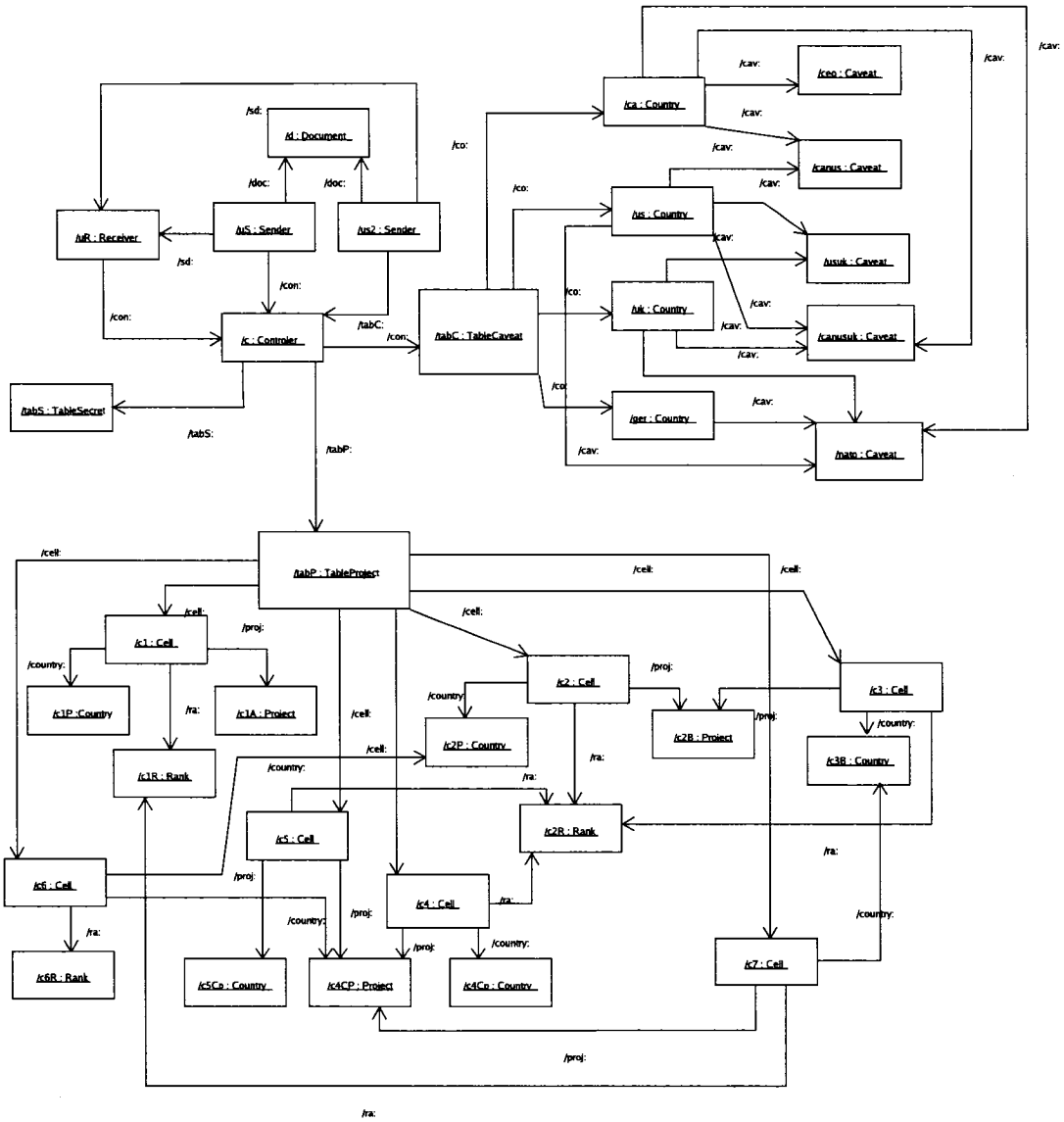

Figure 5.22 Diagramme d'objets - Deux Senders

CHAPITRE 6

CONCLUSION

Dans ce dernier chapitre nous présentons quelques considérations rétrospectives sur ce qui a été réalisé dans ce mémoire. Nous suggérons également quelques directions possibles qui pourraient être poursuivies pour la future recherche en vue d'améliorer l'outil SOCLe.

La modélisation et la vérification orientée-objet peuvent causer plusieurs problèmes. D'une part, dû aux interactions entre les objets, il y a le problème d'explosion combinatoire. D'autre part, les modèles orientés-objet étant des structures de données complexes et difficiles à manipuler, cela réduit la taille des modèles analysables. De plus, la modélisation orientée-objet possède des mécanismes de création dynamique qui provoquent un problème encore plus épineux, celui des espaces d'états infinis. Mais puisque la vérification par model-checking se limite aux systèmes finis, elle est donc quelque peu limitée. Or l'outil SOCLe, dont le but principale de ce travail est l'amélioration de sa performance, permet la validation par model-checking de modèles orientés-objet. La première version de cet outil exige d'une part le calcul complet du graphe d'exécution afin d'appliquer la vérification. D'autre part il utilise l'`ASM` comme formalisme pour décrire la sémantique des modèles. L'`ASM` utilise des structures de données complexes et lourdes en espace mémoire, près des langages de programmation et idéal pour la simulation. Son manque d'abstraction est un handicap à son utilisation pour la vérification. Des stratégies de réduction de l'espace d'états sont donc nécessaires. La stratégie adoptée pour atteindre cet objectif est celle dite *à la volée* qui permet de combattre ces problèmes dans une certaine mesure.

6.1 Réalisations

Un model-checking à la volée requiert un modèle formel, une logique temporelle et un algorithme capable d'arrêter le calcul dès qu'il est en mesure de répondre à la vérification recherchée.

Pour pouvoir vérifier des systèmes qui produisent des graphes d'exécution plus importants, nous avons présenté un modèle formel abstrait qui les représente. Il s'agit d'une structure à base d'objets orientée UML, une abstraction d'ASM, capable de décrire correctement les paradigmes orientés-objet tels que la création d'objets, l'héritage et le polymorphisme indépendamment de tout langage de programmation. Ce modèle est une structure générique orientée-objet pour laquelle SOCLe représente une instance. Il s'agit d'une structure de Kripke étendue où l'information est concentrée dans les états. Un état est complètement défini par une paire de fonctions: une fonction qui décrit les objets et une autre qui décrit les méthodes actives. OCL fait partie d'UML et ce modèle nous permettra d'exprimer la sémantique des expressions OCL, et donc aussi de l'opérateur @*pre*. Pour faciliter la sémantique de ce dernier, une autre fonction est ajoutée. Elle permet de garder un historique pour les instances de méthodes et rend la sémantique de @*pre* triviale.

Après avoir exprimé la sémantique des expressions OCL utilisées dans la modélisation à l'aide de SOCLe, nous avons présenté une logique temporelle orientée-objet permettant d'exprimer une gamme des contrats plus vaste que les standards invariants et pré/post-conditions. Cette logique dénommée OCL^{EXT}, est une extension de la spécification OCL, inspirée par celle de BOTL (Rensink et al., 2002), une logique orientée-objet à deux niveaux. La différence par rapport à BOTL consiste dans le fait que dans le premier niveau on place directement les expressions OCL enrichies par des opérateurs orientés-objet, et dans le deuxième niveau on ajoute les opérateurs temporaux de la logique de CTL, ainsi que des quantificateurs bornés

universel et existentiel. De cette manière ceci rend la spécification OCL plus expressive, en lui permettant d'exprimer des contrats qui spécifient certains aspects du comportement du système en cours d'exécution et soumis à des conditions dynamiques particulières d'interaction. Le choix de CTL est justifié par le fait que cette logique se prête facilement à la vérification à la volée en utilisant une approche basée sur des tableaux. Le problème d'espace infini, qui peut être causé par la création dynamique est résolu en bornant les quantificateurs.

Nous avons également implanté les algorithmes naïf et linéaire à la volée. Ce sont des adaptations des algorithmes de (B. Vergauwen and J. Lewi, 1993) pour le modèle et la logique présentés. Puis, nous avons validé cette technique par une importante étude de cas.

Cette étude de cas ne représente pas un simple exemple. Elle est réalisée pour le compte du RDDC qui a donnée lieu à une publication gouvernementale (Painchaud et al., 2005). Il s'agit de modéliser et de vérifier un système renforcé de sécurité pour les documents d'importance nationale; les *Cavéats*. Les *Cavéats* représentent des *catégories de sécurité*, qui s'additionnent aux restrictions de classifications sécurisées existantes des documents. Elles sont importantes pour le Ministère National de la Défense (Departement of National Defense - DND) et leur Système de Contrôle (Command and Control Information Systems - C2IS). La première partie décrit un moyen de modéliser les données d'entrée à l'aide des objets UML et des expressions OCL. Le comportement dynamique est limité dans cette partie. Le model-checker nous permet de valider cette modélisation. Par la suite un comportement dynamique plus important est ajouté, et on démontre à l'aide de SOCLe que la concurrence mal gérée peut engendrer facilement une fuite d'information. L'étude de cas permet également de mettre en valeur la vérification à la volée. Elle s'avère efficace surtout quand la propriété est vérifiée ou falsifiée sur les préfixes du graphe d'exécution. On constate une réduction du temps de calcul d'une moyenne de 30%.

Pour les cas où on est obligé de parcourir le graphe d'exécution au complet, cette approche s'avère très efficace comparée à la vérification de μ-calcul, quand il s'agit des graphes de grande taille. Enfin une autre remarque s'impose; certaines propriétés s'expriment plus facilement dans cette extension d'OCL qu'avec le μ-calcul (exemple: pour des formules emboîtées de type $[\_U\_]$).

6.2 Travaux futurs

La définition d'un modèle abstrait de calcul basé sur des objets permet d'établir des bases formelles à l'étude du model-checking des systèmes UML. Ce contexte nous a permis d'exprimer la sémantique et l'implantation des algorithmes de vérification à la volées pour la logique OCL^{EXT}. Pour améliorer cette stratégie, il faudrait utiliser l'avantage du model-checking local qui donne la possibilité d'appliquer dans la plupart de cas, une technique d'élagage (*pruning*) à la volée, permettant une gestion judicieuse de l'espace mémoire qui n'influence pas la vérification ultérieure. Cette méthode s'applique quand le graphe est en cours de construction, le résultat de la propriété recherchée est indéterminé et la capacité de la mémoire est dépassée. Pour libérer la mémoire, on emploie la notion de *ramasse miettes*. Cette technique se révèle efficace, compte tenu qu'en pratique les portions de graphe correspondant à l'exécution d'objets sont indépendantes les unes des autres. Nous avons implanté cette technique pour l'algorithme naïf et les tests effectués s'avèrent satisfaisants, mais aucune preuve n'est pour l'instant formalisée.

L'outil SOCLe s'avère être le premier qui présente une sémantique d'un fragment UML et qui permet la vérification exhaustive des contraintes OCL par le model-checking. Par contre, il ne possède pas la robustesse des vérificateurs automatiques de pointe. Les travaux futurs consistent principalement à appliquer des techniques éprouvées de vérification afin d'améliorer la puissance de l'outil. Deux

approches semblent prometteuses permettant de combattre le principal verrou scientifique du model-checking, le phénomène d'explosion combinatoire inhérent à la méthode. La première est l'adaptation des techniques de vérification symboliques afin d'effectuer les vérifications sur une représentation compacte du graphe d'exécution. La deuxième est l'application des méthodes basées sur l'analyse statique des programmes. Par exemple l'analyse statique par *slicing* permet d'extraire automatiquement le modèle à vérifier. Ces deux approches sont tout à fait complémentaires à la méthode de vérification à la volée qui s'avère dans tous les cas efficace.

RÉFÉRENCES

(2003a). OMG Unified Modeling Language specification (version 1.5). Document formal/03-03-01, Object Management Group, Framingham, MA.

B. Vergauwen and J. Lewi (1993). A Linear Local Model Checking Algorithm for CTL. In E. Best, editor, *4th International Conference on Concurrency Theorie (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, Hildesheim, Germany, pages 447–461. Springer-Verlag.

Beer, I., Ben-David, S., Eisner, C., and Geist, D. (1997). RuleBase: Model checking at IBM. *Lecture Notes in Computer Science*, **1254**, 480–483.

Beer, I., Ben-David, S., and Landver, A. (1998). On-the-fly model checking of RCTL formulas. *Lecture Notes in Computer Science*, **1427**, 184–194.

Bergeron, M. (2004). Model-checking UML Designs Using a Temporal Extension of the Object Constraint Language. Master's thesis, École Polytechnique de Montréal.

Bolognesi, T. and Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, **14**(1), 25–59.

Booch, G. (1994). *Object-Oriented Analysis and design.* Addison–Wesley. ISBN: 0–805–35340–2.

Bouali, A., Gnesi, S., and Larosa, S. (1994). The integration project for the JACK environment. Report CS-R9443, CWI, Amsterdam.

Bozga, M., Graf, S., and Mounier, L. (2002). IF-2.0: A validation environment for component-based real-time systems. *Lecture Notes in Computer Science*, **2404**, 343–348.

Bozga, M., Graf, S., Ober, I., Ober, I., and Sifakis, J. (2004). Tools and applications ii: The if toolset. *Lecture Notes in Computer Science*, **3185**, 237–267.

Bradfield, J. and Problem, T. (1998). Observational mu-calculus.

Bradfield, J. C., Stevens, J. C. B. P., Mu-calculus, O., and Stevens, P. (1999). Observational mu-calculus.

Büchi, J. R. (1960). Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundladen der Mathematik*, **6**, 66–92.

Chiorean, D., Bortes, M., Corutiu, D., and Carcu, A. (2001). Object contraint language environment (OCLE).

Chiorean, D., Pasca, M., Carcu, A., Botiza, C., and Moldovan, S. (2004). Ensuring uml models consistency using the ocl environment. *Electr. Notes Theor. Comput. Sci.*, **102**, 99–110.

Clarke, E. M., Browne, M. C., Emerson, E. A., and Sistla, S. (1984). Using temporal logic for automatic verification of finite state systems. In Apt, K. R., editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI*, pages 3–26. Springer-Verlag.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1983). Automatic verification of finite state concurrent systems using temporal logic specifications. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 117–126. ACM, ACM.

Cleaveland, R. (1990). Tableaux-Based Model Checking in the Propositional μ-calculus. *Acta Informatica*, **27**, 725–747.

Distefano, D., Katoen, J., and Rensink, A. (2000). Towards Model Checking OCL. In *Proceedings of ECOOP Workshop on Defining a Precise Semantics for UML, 2000.*

Fernandez, J.-C., Garavel, H., Mounier, L., Rasse, A., Rodríguez, C., Sifakis, J., and Mateescu, R. (1991). Construction and analysis of distributed processes (CADP). http://www.inrialpes.fr/vasy/cadp/.

Fischer, M. J. and Ladner, R. E. (1979, Apr.). Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, **18**(2), 194–211.

Garavel, H., Lang, F., and Mateescu, R. (2002). An overview of cadp 2001. *European Association for Software Science and Technology (EASST) Newsletter*, **4**, 13–24.

Gnesi, S., Latella, D., and Massink, M. (1999). Model checking UML statechart diagrams using JACK. In Paul, R. and Meadows, C., editors, *Proc. of the Fourth IEEE International Symposium on High Assurance Systems Engineering*. IEEE.

Gnesi, S. and Mazzati, F. (2004). On the fly model checking of communication UML state machine. In *Second ACIS International Conference on Software Engineering Research, Management and Applications (SERA2004)*.

Gogolla, M., Bohling, J., and Richters, M. (2003). Validation of UML and OCL models by automatic snapshot generation. In Stevens, P., Whittle, J., and Booch, G., editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 265–279. Springer.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8**(3), 231–274.

Harel, D. and Naamad, A. (1996). The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, **5**(4), 293–333.

Holzmann, G. J. (1993). Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, **25**(9), 981–1017. also in: Proc. 11th PSTV91, INWG/IFIP, Stockholm, Sweden.

Holzmann, G. J. (2003). *The SPIN Model Checker*. Pearson Education.

Jacobsen, I., Christerson, M., Jonsson, P., and Overgaard, G. G. (1992). *Object-Oriented Software Engineering*. Addison-Wesley.

Julian C. Bradfield, Juliana Küster Filipe, P. S. (2002). Enriching ocl using observational mu-calculus. In *Proceedings of FASE2002*. Springer Verlag.

Knapp, A. and Merz, S. (2002). Model checking and code generation for uml state machines and collaborations. In Schellhorn, G. and Reif, W., editors, *FM-TOOLS 2002: 5th Workshop on Tools for System Design and Verification*, Report 2002-11, Reisensburg, Germany. Institut für Informatik, Universität Augsburg.

Knapp, A., Merz, S., and Rauh, C. (2002). Model checking timed UML state machines and collaborations. In Damm, W. and Olderog, E.-R., editors, *7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002)*, volume 2469 of *Lecture Notes in Computer Science*, Oldenburg, Germany, pages 395–414. Springer-Verlag.

Kozen, D. (1983). Results on the propositional mu -calculus. *Theoretical Computer Science*, **27**(3), 333–354.

Kutter, P. and Pierantonio, A. (1997). Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, **3**(5), 416–442.

Latella, D., Majzik, I., and Massink, M. (1999). Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*. Kluwer.

Lilius, J. and Paltor, I. P. (1999). vUML: a tool for verifying UML models. Technical Report TUCS-TR-272, Turku Centre for Computer Science, Finland.

Mateescu, R. and Sighireanu, M. (2003). Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, **46**(3), 255–281.

Nicola, R. D. and Vaandrager, F. W. (1990). Action versus state based logics for transition systems. In Guessarian, I., editor, *Semantics of Systems of Concurrent Processes, Proceedings LITP Spring School on Theoretical Computer Science*, La Roche Posay, France, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag.

Oarga, R. and Bergeron, M. (2004). Automatic Teller Machine: A Case Study. Technical report, Ecole Polytechnique de Montreal.

OCL 1.1 (1997). *Object Constraint Language Specification, version 1.1.* Object Modeling Group.

OMG (2002). Response to the UML 2.0 OCL RfP (ad/2000-09-03). Technical Report ad/2002-05-09, Object Management Group.

OMG (2003b). Unified Modeling Language: Superstructure (ad/2000-09-03). Technical Report OMG RFP ad/00-09-02, Object Management Group.

Painchaud, F., Azambre, D., Bergeron, M., Mullins, J., and R.Oarga (2005). Socle: Integrated design of software applications and security. *10th International Command and Control Research and Technology Symposium (ICCRTS).*

Painchaud, F., Michaud, F., and Charpentier, R. (2004). Caveat Separation: A Case Study. Technical report, Defence Research and Development Canada - Valcartier.

Pnueli, A. (1997). The temporal logic of programs. Technical report.

Porres, I. (2001). *Modeling and Analyzing Software Behavior in UML.* PhD thesis, Turku Centre for Computer Science.

Rensink, A., Distefano, D., and Katoen, J.-P. (2002). On A temporal logic for object-based systems.

Richters, M. and Gogolla, M. (1998). On formalizing the UML object constraint language OCL. *Lecture Notes in Computer Science*, **1507**, 449–464.

Richters, M. and Gogolla, M. (2001). OCL: Syntax, semantics, and tools. *Lecture Notes in Computer Science*, **2263**, 42–68.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modelling and Design.* Prentice-Hall Inc.

S.A. Kripke (1963). Semantical considerations on modal logic. In *Proceedings of a Colloquium: Modal and Many Valued Logics*, volume 16 of *Acta Philosophica Fennica*, pages 83–94.

Stärk, R., Schmid, J., and Börger, E. (2001). *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag.

UML, C. (1997). UML en français. http://uml.free.fr.

van der Beeck, M. (2001). Formalization of UML-statecharts. In Gogolla, M. and Kobryn, C., editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 406–421. Springer.

Wieringa, R. and Broersen, J. (1998). A minimal transition system semantics for lightweight class- and behavior diagrams. In Broy, M., Coleman, D., Maibaum, T. S. E., and Rumpe, B., editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques.* Technische Universität München, TUM-I9803.

ANNEXE I

DÉTAILS SUPPLÉMENTAIRES SUR L'ALGORITHME À LA VOLÉE

Cette annexe expose les détails de la linéarisation d'algorithme à la volée implanté.

I.1 Linéarisation de la procédure *CheckEU*

Le temps quadratique découle de la présence dans une formule à vérifier, des formules imbriquées de type $E[\_U\_]$. Comme départ pour la linéarisation, on prend comme critère le lemme suivant avec certaines notations.

Lemme 1.1 Pour chaque état $s \in \mathcal{S}_{eu}$ on a:

$$[1] \quad (\pi.goal = \bot) \Rightarrow (\mathcal{K}, s \not\models E[\varphi_1 U \varphi_2])$$

$$[2] \quad (\pi.goal \neq \bot) \Rightarrow (\mathcal{K}, s \models E[\varphi_1 U \varphi_2] \ ssi \ \ s \tilde{\rightsquigarrow} \pi.goal)$$

où

- $\mathcal{S}_{eu}$ l'ensemble d'états marqués par $CheckEU(s_r, \varphi_1, \varphi_2)$.
- $\rightsquigarrow$ l'ensemble de transitions marquées par $CheckEU(s_r, \varphi_1, \varphi_2)$.
- $\tilde{\rightsquigarrow}$ est la fermeture réflexive, transitive de $\rightsquigarrow$.
- $\pi.goal$ la valeur de la variable *goal* dans le point π.

Il reste à trouver un algorithme efficace pour calculer l'ensemble:

$$\{s \in \mathcal{S}_{eu} \mid s \tilde{\rightsquigarrow} \pi.goal\} \text{ en supposant que } \pi.goal \neq \perp \tag{I.1}$$

La façon la plus simple pour calculer cet ensemble est par chaînage arrière à partir de $\pi.goal$ en utilisant la fonction inverse de la relation de transitions $\rightsquigarrow$, comme elle est présentée en (Clarke et al., 1984). Cette approche présente l'inconvénient de stocker non seulement la transition mais également son inverse. Pour éliminer ce handicap, on essaie d'inclure le calcul de l'ensemble (I.1) dans la recherche en profondeur, exécutée par *CheckEU*. On utilise une variable qui cumule de l'information sur cet ensemble à chaque état marqué.

Notations:

- $<$ un opérateur binaire qui établit une relation d'ordre entre deux états, comme suit:
  $\forall s, s' \in \mathcal{S}_{eu} : s < s'$ ssi l'état s a été marqué avant s'.

- Soit $F : \mathcal{S}_{eu} \rightarrow \mathcal{P}(\mathcal{S}_{eu})$:

  $$F(s) = \{s' \leq s \mid s' \tilde{\rightsquigarrow} s\}$$

  Elle représente tous les prédécesseurs de l'état s.
  Puisque $\pi.goal$ est l'élément maximal (l'état) qui correspond à l'ordre $<$, on a:

  $$\{s \in \mathcal{S}_{eu} \mid s \tilde{\rightsquigarrow} \pi.goal\} = F(\pi.goal)$$

  Donc, il suffit de trouver un algorithme pour la fonction F.

Exemple 1.1 Soit le système de transitions figure I.1 et la formule $\phi = E[\text{True } U\ p]$.

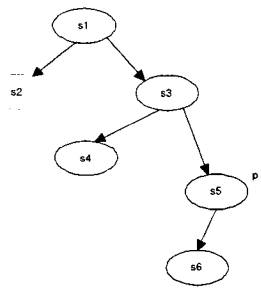

Figure I.1 Graphe

L'ensemble d'états marqués par $CheckEU$:

$\mathcal{S}_{eu} = \{s_1, s_2, s_3, s_4, s_5\}$

La fonction F pour chaque état visité:

$F(s1) = \{s_1\}$

$F(s2) = \{s_1, s_2\}$

$F(s3) = \{s_1, s_3\}$

$F(s4) = \{s_1, s_3, s_4\}$

$F(s5) = \{s_1, s_3, s_5\}$ dans $s5$ ϕ est vraie

$\pi.goal = s_5$ et $F(s_5) = \{s \in \mathcal{S}_{eu} \mid s \tilde{\rightsquigarrow} s_5\}$

Calcul de la fonction F. Le but est d'obtenir une caractérisation récursive pour la fonction F qui permet de calculer F en parcourant une seule fois le système de transition. La technique de recherche est toujours *en profondeur* dans le dépliage du graphe.

Notations:

- Les noeuds de cet arbre sont les états de l'ensemble $\mathcal{S}_{eu}$.
- Soit $\rightarrow$ un ensemble d'arcs tels que: $\rightarrow \in \rightsquigarrow$. Si $s \rightarrow s'$, alors,

- s' est un *enfant* de s
- s est le *père* de s' (la racine d'arbre s_r n'a pas de père)

• Soit $\mathcal{S}^r_{eu} = \mathcal{S}_{eu} \setminus \{s_r\}$. Pour chaque $s \in \mathcal{S}^r_{eu}$, on dénote:

- $s.fa$, le père de s
- $s.sons$, les fils de s
- $s.left$, l'ensemble des frères à gauche de s, i.e.,
  $s.left = \{s' \in \mathcal{S}^r_{eu} \mid s' < s \quad \text{et} \quad s'.fa = s.fa\}$

On commence par la racine s_r et on a évidement: $F(s_r) = \{s_r\}$

Pour un $s \neq s_r$ on a:

$$F(s) = \{s' \leq s \mid s' \overset{\sim}{\rightsquigarrow} s\} = \{s\} \cup \{s' < s \mid s' \overset{\sim}{\rightsquigarrow} s\} = \{s\} \cup \{s' \leq s \mid s' \overset{\sim}{\rightsquigarrow} s.fa\}$$
$$= \{s\} \cup \{s' \leq s.fa \mid s' \overset{\sim}{\rightsquigarrow} s.fa\} \cup \bigcup_{s' \in s.left} \{s'' \mid s' \overset{\sim}{\rightarrow} s'' \overset{\sim}{\rightsquigarrow} s.fa\}$$
$$= \{s\} \cup F(s.fa) \cup \bigcup_{s' \in s.left} \{s'' \mid s' \overset{\sim}{\rightarrow} s'' \overset{\sim}{\rightsquigarrow} s'.fa\}$$

Exemple 1.2 Soit le graphe de la figure I.2.

Soit:

$s = s_8$

$s.fa = \{s_5\}$

$s.sons = \{s_{13},\ s_{14}\}$

$s.left = \{s_6,\ s_7\}$

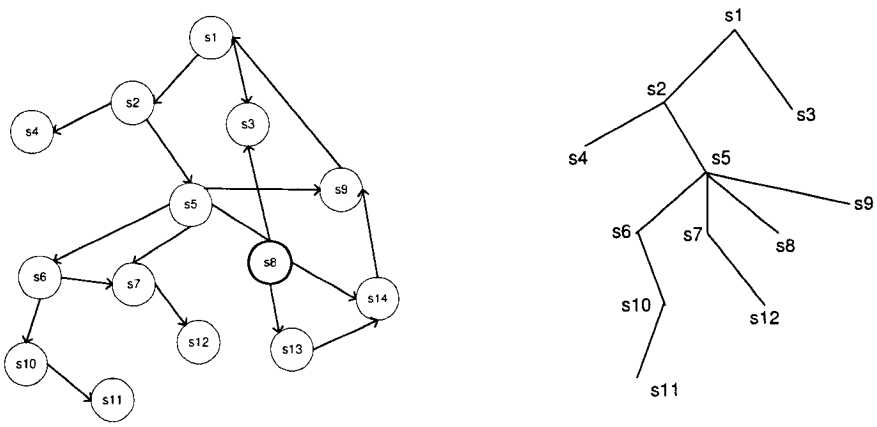

Figure I.2 Linéarisation - Étape 1

$$\mathcal{S}_{eu} = \{s_1,\ s_2,\ s_3,\ s_4,\ s_5,\ s_6,\ s_7,\ s_8,\ s_9,\ s_{10},\ s_{11},\ s_{12}\}$$

$$\begin{aligned}
F(s_8) &= \{s_1,\ s_2,\ s_5,\ s_6,\ s_{10},\ s_{11},\ s_8,\ s_7,\ s_{12}\} \\
&= \{s_8\} \cup \{s_1,\ s_2,\ s_5,\ s_6,\ s_{10},\ s_{11},\ s_7,\ s_{12}\} \\
&= \{s' < s_8 \mid s' \tilde{\rightsquigarrow} s_8\} \\
&= \{s_8\} \cup \{s' < s_8 \mid s' \tilde{\rightsquigarrow} s_8.fa\} \\
&= \{s_8\} \cup \{s_1,\ s_2,\ s_5\} \cup \bigcup_{s'' \in \{s_6,\ s_7\}} \{\{s_7,\ s_{12}\}, \{s_6,\ s_{10},\ s_{11}\}\} \\
&= \{s_8\} \cup \{s' < s_8.fa \mid s' \tilde{\rightsquigarrow} s_8.fa\} \cup \bigcup_{s'' \in s_8.left} \{s'' \mid s' \tilde{\rightarrow} s'' \tilde{\rightsquigarrow} s_8.fa\} \\
&= \{s_8\} \cup F(s_8.fa) \cup \bigcup_{s'' \in s_8.left} \{s'' \mid s' \tilde{\rightarrow} s'' \tilde{\rightsquigarrow} s_8.fa\}
\end{aligned}$$

Soit une fonction: $\Delta : \mathcal{S}_{eu}^{r} \rightarrow \mathcal{P}(\mathcal{S}_{eu})$:

$$\Delta(s) = \{s' \mid s \tilde{\rightarrow} s' \tilde{\rightsquigarrow} s.fa\} \tag{I.2}$$

La fonction F prend la forme récursive suivante:

$$F(s_r) = \{s_r\}$$
$$(s \neq s_r) \Rightarrow F(s) = \{s\} \cup F(s.fa) \cup \bigcup_{s' \in s.left} \Delta(s')$$

Il reste maintenant à exprimer Δ aussi d'une façon récursive.

$$\begin{aligned}
\Delta(s) &= \{s' \mid s \overset{\cdot}{\rightarrow} s' \overset{\cdot}{\rightsquigarrow} s.fa\} = \{s' \mid s \overset{\cdot}{\rightarrow} s' \overset{\cdot}{\rightsquigarrow} s \overset{\cdot}{\rightsquigarrow} s.fa\} \\
&= \texttt{if } s \overset{\cdot}{\rightsquigarrow} s.fa \texttt{ then } \{s' \mid s \overset{\cdot}{\rightarrow} s' \overset{\cdot}{\rightsquigarrow} s\} \texttt{ else } \varnothing \\
&= \texttt{if } s \overset{\cdot}{\rightsquigarrow} s.fa \texttt{ then } \{s\} \cup \bigcup_{t \in s.sons} \{s' \mid t \overset{\cdot}{\rightarrow} s' \overset{\cdot}{\rightsquigarrow} s\} \texttt{ else } \varnothing \\
&= \texttt{if } s \overset{\cdot}{\rightsquigarrow} s.fa \texttt{ then} \{s\} \cup \bigcup_{t \in s.sons} \Delta(t) \texttt{ else } \varnothing \\
&= \texttt{if } \exists s' < s \mid s \overset{\cdot}{\rightarrow} \rightsquigarrow s' \overset{\cdot}{\rightsquigarrow} s.fa \texttt{ then } \{s\} \cup \bigcup_{t \in s.sons} \Delta(t) \texttt{ else } \varnothing \\
&= \texttt{if } \exists s' < s \mid s \overset{\cdot}{\rightarrow} \rightsquigarrow s' \overset{\cdot}{\rightsquigarrow} s \texttt{ then } \{s\} \cup \bigcup_{t \in s.sons} \Delta(t) \texttt{ else } \varnothing \\
&= \texttt{if } \exists s' < s \mid s' \in F(s) \text{ et } s \overset{\cdot}{\rightarrow} \rightsquigarrow s' \texttt{ then } \{s\} \cup \bigcup_{t \in s.son} \Delta(t) \texttt{ else } \varnothing
\end{aligned}$$

Note 1.1 $s \overset{\cdot}{\rightarrow} \rightsquigarrow s'$ est un raccourci de $\exists s'' \mid s \overset{\cdot}{\rightarrow} s'' \rightsquigarrow s'$.

Soit la fonction $G : \mathcal{S}^r_{eu} \rightarrow \mathcal{P}(\mathcal{S}_{eu})$:

$$G(s) = \{s' < s \mid s' \in F(s) \text{ et } s \overset{\cdot}{\rightarrow} \rightsquigarrow s'\} \tag{I.3}$$

Maintenant on peut exprimer la fonction Δ:

$$\Delta(s) = \texttt{if } G(s) \neq \varnothing \texttt{ then } \{s\} \cup \bigcup_{t \in s.sons} \Delta(t) \texttt{ else } \varnothing$$

Finalement on exprime aussi récursivement la fonction G:

$$\begin{aligned} G(s) &= \{s' < s \mid s' \in F(s) \text{ et } s \tilde{\rightarrow} \rightsquigarrow s'\} \\ &= \{s' < s \mid s' \in F(s) \text{ et } s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \{s' < s \mid s' \in F(s) \text{ et } t \tilde{\rightarrow} \rightsquigarrow s'\} \\ &= \{s' < s \mid s' \in F(s) \text{ et } s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \{s' < s \mid s' \in F(t) \text{ et } t \tilde{\rightarrow} \rightsquigarrow s'\} \\ &= \{s' < s \mid s' \in F(s) \text{ et } s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \{s' < s \mid s' \in G(t)\} \end{aligned}$$

Donc:

$$G(s) = \{s' < s \mid s' \in F(s) \text{ et } s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \{s' < s \mid s' \in G(t)\} \qquad \text{(I.4)}$$

Finalement on a les trois fonctions F, Δ, G:

[1] $F(s_r) = \{s_r\}$

[2] $(s \neq s_r) \Rightarrow F(s) = \{s\} \cup F(s.fa) \cup \bigcup_{s' \in s.left} \Delta(s')$

[3] $G(s) = \{s' < s \mid s' \in F(s) \text{ et } s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \{s' < s \mid s' \in G(t)\}$

[4] $\Delta(s) = \texttt{if } G(s) \neq \varnothing \texttt{ then } \{s\} \cup \bigcup_{t \in s.sons} \Delta(t) \texttt{ else } \varnothing$

Plus formellement, comme $s' < s$ pour $\forall s' \in G(s)$, on a:

$$G(s) = \varnothing \;\Leftrightarrow\; Min(\{s\} \cup G(s)) = s \qquad \text{(I.5)}$$

où $Min(Q)$ dénote le plus petit élément de Q en respectant la définition de $<$, $(Q \neq \varnothing)$. En plus:

$$
\begin{aligned}
&Min(\{s\} \cup G(s)) \\
&\qquad = Min(\{s\} \cup \{s' < s \mid s' \in F(s) \text{ et } s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \\
&\qquad\quad \{s' < s \mid s' \in G(t)\}) \\
&\qquad = Min(\{s\} \cup \{s' < s \mid s' \in F(s) \text{ et } s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} G(t)) \\
&\qquad = Min(\{s\} \cup \{s' < s \mid s' \in F(s) \text{ et } s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \\
&\qquad\quad (\{t\} \cup G(t))) \\
&\qquad = Min(\{s\} \cup \{s' < s \mid s' \in F(s) \text{ et } s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \\
&\qquad\quad \{Min(\{t\} \cup G(t))\})
\end{aligned}
$$

Soit la fonction $H : \mathcal{S}^r_{eu} \rightarrow \mathcal{S}_{eu}$

$$H(s) = Min(\{s\} \cup G(s))$$

On obtient l'équation suivante:

$$H(s) = Min(\{s\} \cup \{s' < s \mid s' \in F(s) \text{et} s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \{H(t)\}) \qquad \text{(I.6)}$$

Maintenant on peut remplacer l'équation [3]:

$$[1]\ F(s_r) = \{s_r\}$$

$$[2]\ (s \neq s_r) \Rightarrow F(s) = \{s\} \cup F(s.fa) \cup \bigcup_{s' \in s.left} \Delta(s')$$

$$[3]\ H(s) = Min(\{s\} \cup \{s' < s \mid s' \in F(s) \text{et} s \rightsquigarrow s'\} \cup \bigcup_{t \in s.sons} \{H(t)\})$$

$$[4]\ \Delta(s) = \texttt{if}\ H(s) \neq s\ \texttt{then}\ \{s\} \cup \bigcup_{t \in s.sons} \Delta(t)\ \texttt{else}\ \varnothing$$

ANNEXE II

DÉTAILS DE L'ÉTUDE DE CAS

Cette annexe contient dans la première section les détails de l'étude de cas présenté au chapitre 5. Plus précisément, chaque tableau expose les transitions formelles d'un diagramme états-transitions. Dans la deuxième section quelques résultats obtenus sur l'étude de cas de guichet automatique serons présentés, permettant de sortir un évidence l'efficacité de la vérification à la volée et surtout de la linéarisation.

II.1 Les transitions formelles

Les fichiers `UML` correspondants à chaque étape de la modélisation:

- Étape 1 - Modélisation du tableau Classification: *caveatEtape1.zargo.*
- Étape 2: - Modélisation des tableaux Classification et Caveat: *caveatEtape2.zargo.*
- Étape 3 et 4: - Modélisation des tableau Classification, Caveat et Projet: *caveatEtape34.zargo.*
- Étape 5: Contraintes (tableaux) et comportement:*caveatC.zargo.*
- Étape 6: Un usager de plus (explosion combinatoire):*caveat2sender.zargo.*

Tableau II.1 Symboles formels sur une transition

Déclencheur absent	$\star$
Pas de garde	True
Pas d'action	-
Retoure méthode	*%nom méthode*

Tableau II.2 Transitions Sender - *AskSend*

s5

trigger : $\star$

guard : True

action : $self$.User/con.Controller/$accessSend(u1$:User, $u2$:User, d : Document) :Bool[$self$, $head(self$.Sender/$sd)$, $head(self$.User/$doc)$];Sender/$accessRead$ =False

s6

trigger : Controller%$accessSend(u1$:User, $u2$:User, d :Document) :Bool

guard : $r\_accessSend$ = True

action : Sender/$accessSend$ = True;
$self$.Sender/sd.Receiver/$receive(d$: Document) :
Bool[$head(self$.User/$doc)$]

s7

trigger : Controller%$accesSend(u1$:User, $u2$:User, d :Document) : Bool

guard : $r\_accesSend$ = False

action : Sender/$accesSend$ = False

Tableau II.3 Transitions `Controller` - *AskAccess*

c1
trigger : `Controller`/*access*(*u* :`User`, *d* :`Document`) :Bool
guard : True
action : *self*.`Controller`/*tabS*.`Table`/*accessTable*(*u* :`User`, *d* :`Document`) : Bool[u, d]

c2
trigger : `Table`%*accessTable*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_accessTable* = True
action : *self*.`Controller`/*tabC*.*Table*/*accessTable*(*u* :`User`, *d* :`Document`) : Bool[u, d]

c3
trigger : `Table`%*accessTable*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_accessTable* = True
action : *self*.`Controller`/*tabP*.`Table`/*accessTable*(*u* :`User`, *d* :`Document`) : Bool[u, d]

c4
trigger : `Table`%*accessTable*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_accessTable* = True
action : `Controller`%*access*(*u* : `User`, *d* :`Document`) :Bool[True]

c5
trigger : `Table`%*accessTable*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_accessTable* = False
action : `Controller`%*access*(*u* : `User`, *d* :`Document`) :Bool[False]

c6
trigger : `Table`%*accessTable*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_accessTable* =False
action : `Controller`%*access*(*u* : `User`, *d* :`Document`) :Bool[False]

c7
trigger : `Table`%*accessTable*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_accessTable* = False
action : `Controller`%*access*(*u* : `User`, *d* :`Document`) :Bool[False]

Tableau II.4 Transitions `Controller` - *AskSend* et *ChangeTable*

c9
trigger : `Controller`/*accessSend*(*u*1 :`User`, *u*2 :`User`, *d* :`Document`) :Bool
guard : true
action : *self* ::null.`Controller`/*access*(*u* :`User`, *d* :`Document`) :Bool[u1,d]

c8
trigger : `Controller`%*access*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_access* = True
action : *self* ::null.`Controller`/*access*(*u* :`User`, *d* :`Document`) :Bool[u2,d]

c10
trigger : `Controller`%*access*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_access* = True
action : `Controller`%*accessSend*(*u*1 :`User`, *u*2 : `User`, *d* :`Document`) : Bool[True]

c11
trigger : `Controller`%*access*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_access* = False
action : `Controller`%*accessSend*(*u*1 :`User`, *u*2 : `User`, *d* :`Document`) : Bool[False]

c12
trigger : `Controller`%*access*(*u* :`User`, *d* :`Document`) :Bool
guard : *r_access* = False
action : `Controller`%*accessSend*(*u*1 :`User`, *u*2 : `User`, *d* :`Document`) : Bool[False]

c14
trigger : ⋆
guard : True
action : *self*.`Controller`/*tabP*.`Table`/*updateTable*() :void[]

Tableau II.5 Transitions `TableSecret` - *TabSecretStatechart*

ts1	
trigger :	`TableSecret`/$accessTable$(u :`User`, d :`Document`) :Bool
guard :	u.`User`/$clearance$ >= d.`Document`/$level$
action :	`TableSecret`%$accessTable$(u :`User`, d :`Document`) :Bool[True]
ts2	
trigger :	`TableSecret`/$accessTable$(u :`User`, d :`Document`) :Bool
guard :	!(u.`User`/$clearance$ >= d.`Document`/$level$)
action :	`TableSecret`%$accessTable$(u :`User`, d :`Document`) :Bool[False]

Tableau II.6 Transitions `TableCaveat` - *TableCaveatStatechart*

tc1	
trigger :	`TableCaveat`/$accessTable$(u :`User`, d :`Document`) :Bool
guard :	((($self$.`TableCaveat`/co).filter{x | x.`Country`/$codeP$ = u.`User`/$country$}).`Country`/cav). exists{y | y.`Caveat`/$codeC$ = d.`Document`/$caveat$}
action :	`TableCaveat`%$accessTable$(u :`User`, d :`Document`) :Bool[True]
tc2	
trigger :	`TableCaveat`/$accessTable$(u :`User`, d :`Document`) :Bool
guard :	!((((($self$.`TableCaveat`/co).filter{x | x.`Country`/$codeP$ = u.`User`/$country$}).`Country`/cav). exists{y | y.`Caveat`/$codeC$ = d.`Document`/$caveat$}))
action :	`TableCaveat`%$accessTable$(u :`User`, d :`Document`) :Bool[False]

Tableau II.7 Transitions `TableProject` - *TableProjectStatechart*

tp1

trigger : `TableProject`/$accessTable(u$:`User`, d :`Document`) :Bool

guard : $(((self$.`TableProject`/$cell)$.filter$\{x \mid((x$.`Cell`/$country)$.1).
`Country`/$codeCt = u$.`User`/$country\}$).
filter$\{y \mid((y$.`Cell`/$proj)$.1).`Project`/$codePj = d$.`Document`/$projet\}$).
exists$\{z \mid((z$.`Cell`/$ra)$.1).`Rang`/$codeR = u$.`User`/$rank\}$

action : `TableProject`$\%accessTable(u$:`User`, d :`Document`) :Bool[True]

tp2

trigger : `TableProject`/$accessTable(u$:`User`, d :`Document`) :Bool

guard : !((((self.`TableProject`/$cell)$.filter$\{x \mid((x$.`Cell`/$country)$.1).
`Country`/$codeCt = u$.`User`/$country\}$).
filter$\{y \mid((y$.`Cell`/$proj)$.1).`Project`/$codePj = d$.`Document`/$projet\}$).
exists$\{z \mid((z$.`Cell`/$ra)$.1).`Rang`/$codeR = u$.`User`/$rank\}$)

action : `TableProject`$\%accessTable(u$:`User`, d :`Document`) :Bool[False]

tp3

trigger : `TableProject`/$updateTable()$:void

guard : True

action : `TableProject`$\%updateTable()$;
$((self$.`TableProject`/$cell)$.2) ::null.`Cell`/$changeRank(r$:`Rang`) :
void[$((self$.`TableProject`/$cell)$.`Cell`/$ra)$.1]

II.2 Résultats sur l'étude de cas d'ATM

Un guichet automatique (Automatic Teller Machine ATM) est un dispositif électronique qui permet aux clients d'une banque de faire des retraits en espèce et de vérifier leur compte à tout moment sans avoir à consulter un caissier. Beaucoup d'ATMs permettent également aux clients de déposer de l'argent comptant ou des chèques et de transférer de l'argent entre leurs comptes bancaires.

Cette étude de cas a été présenté en détail dans le rapport technique du projet SOCLe,(Oarga and Bergeron, 2004). Elle modélise strictement la fonctionnalité principale d'un ATM, le retrait d'argent.

Obtenir de l'argent suppose que l'usager:

- utilise une carte valide;
- s'identifie par un numéro d'identification personnel (*pin*) correct;
- ne dépasse pas trois essais (trois fautes de frappe permises).

Tableau II.8 Résultats de vérifications

Tech./Prop.	ϕ_1 - False	ϕ_2 - True	ϕ_3 - False
μ-calculus	0.53 / 224	0.45 / 224	–
OCL^{EXT} - naïf	0.01/7	0.37 / 224	1.57/224
OCL^{EXT} - linéaire	0.01/7	0.43 / 224	0.55/224
	temps(s) / nb. d'états		

Le tableau II.8 présente les résultats obtenus pour trois invariants. Le premier, ϕ_1, exprime que jamais un *pin* incorrect n'arrive dans le système. Cet invariant est faut. Le resultat est obtenu après la construction complète du graphe, 224 états et 0.53s, avec l'algorithme de μ-calcul. En utilisant l'algorithme à la volée, après 7 états et en temps de 0.01s on a déjà la réponse. Cet invariant sort en évidence

l'efficacité de la vérification à la volée. Le deuxième invariant, ϕ_2, exprime que si l'usager reçoit de l'argent, alors le nombre d'essais est inférieur à trois. Comme l'invariant est vrai, le graphe d'exécution est construit au complet que ça soit par le μ-calcul ou à la volée. Pour des cas pareils on ne réalise pas un gain de temps en utilisant la vérification à la volée.

La troisième vérification met en valeur l'efficacité de l'algorithme linéaire. Plus précisément cet invariant exprimé par la formule $\phi_3 = E[\mathsf{True}\ U\ E[\mathsf{True}\ U\ \mathsf{False}]]$, nécessite un temps quadratique en utilisant l'algorithme naïf (voir chapitre 4, section 4.6). Par contre avec l'algorithme linéaire on réduit le temps de calcule d'un facteur trois. Donc, il est indiqué d'employer l'algorithme linéaire pour la vérification des formules embriquées de type $E[\_U\_]$, qui provoquent la non-linéarité. De plus il faut souligner qu'exprimer des telles formules dans la logique de μ-calcul s'avère difficile.